Hardware/Software Deadlock Avoidance for Multiprocessor Multiresource System-on-a-Chip

A Thesis
Presented to
The Academic Faculty

by

Jaehwan Lee

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

School of Electrical and Computer Engineering
Georgia Institute of Technology
November 2004

Hardware/Software Deadlock Avoidance for Multiprocessor Multiresource System-on-a-Chip

Approved by:

Professor Vincent J. Mooney III, Adviser

Professor William D. Hunt

Professor Douglas M. Blough
(Reading Committee)

Professor Panagiotis Manolios
(College of Computing)

Professor Sung Kyu Lim
(Reading Committee)

Date Approved: 12 November 2004

To my wife, Young-Hee Yun,

and

my son, Dongyun James Lee,

for their love, support and sacrifices.

PREFACE

It has been four years and a few more months since I started pursuing my Ph.D. These years have been my very precious time of experiencing not only new worlds but also state-of-the-art technology. At the same time, I have suffered a lot from my lack of knowledge and forgetfulness, lack of fluent language communication ability while learning new concepts and describing papers well enough to be published. Now, as my time of graduation nears, I have been wondering if my work is sufficient to uphold the honor of Georgia Tech and also to enable me to find a good job in the United States. I did not have enough time to make this thesis perfect according to my criteria because I needed to write it as soon as possible so that a draft version could be read by proof readers who promised to help me, and so this work could be timely delivered to committee members who would give me feedback as well as criticism.

During my defense, I was asked numerous questions and was given many constructive comments. Feeling the warmth of the committee members, I tried to accommodate all their questions and concerns and then added almost all answers and comments to them in this thesis. In fact, I devised another approach to resolving a priority inversion type of problem in one of our approaches, implemented it in hardware, simulated it and added it to this thesis.

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude and appreciation to everyone who made this thesis possible. First and foremost, I would like to thank my adviser, Dr. Vincent John Mooney III, for his patience and guidance during my Ph.D. study at Georgia Tech. With his knowledge and experience, he has guided me to successfully achieve my research objective. I truly admire him for all his encouragement throughout this process. He was gracious to me and always there when I needed him.

Second, I would like to thank my thesis committee members, Dr. Douglas M. Blough, Dr. William D. Hunt, Dr. Sung Kyu Lim, Dr. Panagiotis Manolios and Dr. Vincent Mooney for their critical evaluation and valuable suggestions, especially Dr. Blough, Dr. Lim and Dr. Vincent Mooney for their thorough reading and fine tuning of my thesis.

Third, I would like to thank all members of the Hardware/Software Codesign Group at Georgia Tech for their friendship, support and feedback, especially, Bilge S. Akgul who helped me adjust to new research as well as to my adviser during my early period at Georgia Tech. Ever since becoming coauthors of a conference paper, we have helped each other with much critical information in the design and simulation of multiprocessor systems.

Fourth, I also thank my family for their love and support throughout my entire Ph.D. study. Without my beloved wife Young-Hee, I would not have even dreamed of a Ph.D., nor would I have finished this Ph.D. My wife waited for me at night, prepared everything that I could not do by myself, endured difficult times with prayer and continually encouraged me to accomplish my vision. Since I did not have much chance to spend time with my son James, he missed me numerous days, which made me feel sorry. But, he also has encouraged me by doing his duty well and growing up well and righteously.

Last, but not least, special thanks to God, who has inspired me to accomplish this research successfully.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

The main objective of this thesis is to implement fast and deterministic hardware/software deadlock avoidance by a novel scalable hardware technique that is easily applicable to real-time multiresource MultiProcessor System-on-a-Chip (MPSoC) design.

Our solutions are provided in the form of Intellectual Property (IP) hardware units which we call the Deadlock Avoidance Unit (DAU) and the Parallel Banker's Algorithm Unit (PBAU).

A novel Parallel Deadlock Detection Algorithm (PDDA) and its hardware implementation in the Deadlock Detection Unit (DDU) were first proposed by Shui, Tan and Mooney. The DDU performs very fast deadlock detection since it traces neither cycles nor paths, nor does it require linked lists.

Our main contributions regarding the DDU are detailed descriptions of PDDA and the DDU with mathematical representations, software implementations of PDDA, a proof of the correctness of PDDA, a proof of the run-time complexity of the DDU, and extensive experimentation among the DDU, PDDA in software and a comparable $O(m \times n)$ deadlock detection algorithm. Our proof of the correctness of PDDA utilizes five lemmas and four theorems; our proof of DDU complexity shows a worst case run-time of $2 \times min(m, n) - 3 = O(min(m, n))$ (where m and n are the numbers of resources and processes, respectively) utilizing two corollaries, one lemma and one theorem. Previous deadlock detection algorithms in software, by contrast, have an $O(m \times n)$ run-time complexity. The DDU reduces deadlock detection time by 99%, (i.e., 100X) or more compared to software implementations of deadlock detection algorithms. An experiment involving a practical situation that employs the DDU showed that the time measured from application initialization to deadlock detection was reduced by 46% compared to detecting deadlock in

software.

The DAU, the second hardware solution, provides very fast and automatic deadlock avoidance in MPSoC with multiple processors and multiple resources. The DAU avoids deadlock by not allowing any grant or request that leads to a deadlock. In case of livelock, the DAU asks one of the processes involved in the livelock to release resource(s) so that the livelock can also be resolved. We devised four novel deadlock avoidance algorithms, implemented the algorithms in Verilog Hardware Description Language (HDL) and synthesized them using an automatic synthesis tool. We simulated two synthetic applications that can benefit from the DAU and demonstrated that the DAU not only avoids deadlock in a few clock cycles but also achieves in our examples approximately 40% speedup of application execution time over avoiding deadlock in software. The MPSoC area overhead due to the DAU is small, under 0.04% in our SoC example.

While the DAU provides automatic deadlock avoidance for single-instance resource systems, Parallel Banker's Algorithm Unit (PBAU), a hardware implementation of our novel Parallel Banker's Algorithm (PBA), accomplishes fast, automatic deadlock avoidance for multiple-instance resource systems. PBA is a parallelized version of the Banker's Algorithm proposed by Habermann for a multiple instance multiple resource system. We have implemented PBA in Verilog HDL and synthesized it using an automatic synthesis tool. PBAU provides a system with an $O(n)$ run-time complexity deadlock avoidance with a best case run-time of $O(1)$. We demonstrate that PBAU not only avoids deadlock in a few clock cycles (1600X faster than the Banker's Algorithm implemented in software) but also achieves in a particular example a 19% speedup of application execution time over avoiding deadlock in software. The MPSoC area overhead due to PBAU is small, under 0.05% in our candidate MPSoC example.

To automate the design of hardware deadlock solutions, we also provide an automatic deadlock hardware generation tool that is capable of generating a custom DDU, DAU or PBAU for a user-specified combination of resources and processes, so that users can easily

and rapidly implement a particular deadlock hardware solution for their target MPSoCs.

Finally, we have integrated automatic generation of DDU, DAU and PBAU into the δ hardware/software Real-Time Operating System (RTOS) partitioning framework, the goal of which is to speed up RTOS/MPSoC codesign. The δ framework is specifically designed to help RTOS/MPSoC designers very easily and quickly explore the available design space with different hardware and software modules so that they can efficiently search and discover several compact solutions matched to the specifications and requirements of their design prior to any actual implementation. We also describe an approach to an automatic deadlock hardware generation tool used to customize a deadlock hardware IP for a particular target system.

CHAPTER I

INTRODUCTION

1.1 Problem Statement

Current trends show that System-on-a-Chip (SoC) technology has contributed to a significant evolution in digital chip design. Unlike a Printed-Circuit-Board (PCB) filled with many digital chips, an SoC is designed as a hardware platform that integrates most of the functions of the end product in a single chip. An SoC has commonly incorporated at least one Processing Element (PE) (e.g., a microprocessor or a Digital Signal Processing Processor (DSP)) or more that run embedded software. An SoC may include peripherals, reconfigurable logic and interfaces to the outside world, and it typically employs a bus-based architecture. An SoC may also contain both memory and analog functions. Current and future SoC technology will facilitate the creation of complex digital systems that are small, portable, energy efficient and reliable. Some examples of such complex digital systems are miniature Personal Digital Assistants (PDAs) and digital cameras.

Current trends also pack more and more data streaming applications with many processes and resources on a single SoC. Thus, as SoC integration accelerates, many more interactions among processes and resources occur. Moreover, many applications consist of processes that require exclusive accesses not just to one hardware resource (e.g., a custom FFT unit), but to several resources, increasing the likelihood of deadlock. Thus, we predict that there will be a significant need for deadlock detection as well as avoidance in an SoC.

Furthermore, the users of a Real-Rime Operating System (RTOS) desire predictable response time at an affordable cost. To fulfill this, many researchers have investigated various approaches to ensure RTOS predictability. One active approach is to utilize one or more hardware mechanism(s) since (i) hardware is typically far more predictable than a software

implementation of the same algorithm, and (ii) the cost of hardware decreases dramatically in accordance with Moore's law (or, more accurately, Moore's prediction) [21].

All deadlock detection or avoidance algorithms known to date have a run-time complexity of at least $O(m \times n)$ (where m is the number of resources and n is the number of processes) since they assume an execution paradigm of one instruction or operation at a time. With a custom hardware implementation of a deadlock algorithm, parallelism can be exploited, thereby reducing run-time dramatically. The objective of this research is to implement deadlock avoidance utilizing hardware mechanisms that are easily applicable to shared-memory multiprocessor SoC design.

Detection of deadlock is extremely important since any request for or grant of a resource might result in deadlock. Invoking software deadlock detection on every resource allocation event would cost too much computational power; thus, using a software implementation of deadlock detection would perhaps be impractical in terms of the performance cost. A promising way of enabling deadlock detection with small computational power is to implement deadlock detection and/or avoidance in hardware. In fact, without the aid of hardware, deadlock checking and meeting every other deadline would be unthinkable for most real-time embedded systems that need to detect and avoid deadlock. With hardware support, however, practical systems may potentially initiate deadlock recovery more quickly and save large investments. A real-life example of a large investment that was almost lost is the Mars Pathfinder, a real-time robot, which had to reset due to a priority inversion condition; new code was downloaded over radio [40]. By quickly detecting such a situation and calling a special "recover" boot code (which was proven by hand to have no deadlock), accidents can potentially be avoided.

1.2 Contributions

This thesis mainly presents two hardware solutions, and necessary proofs, to deadlock problems in MultiProcessor Systems-on-a-Chip (MPSoC). The following items are the

main contributions of this research:

- Proof of the correctness of the Parallel Deadlock Detection Algorithm (PDDA).
- Proof of the run-time complexity of the Deadlock Detection Unit (DDU).
- Design of a novel Deadlock Avoidance Algorithm and its hardware implementation, the Deadlock Avoidance Unit (DAU).
- Proof of the correctness of the Deadlock Avoidance Algorithm.
- Design of a novel Parallel Banker's Algorithm (PBA) and its hardware implementation, the Parallel Banker's Algorithm Unit (PBAU).
- Automatic generation of hardware solutions for deadlock and integration of these solutions into the δ Hardware/Software RTOS partitioning framework.

1.3 Terminology

In this section, we define terms used in this thesis, give examples of deadlock, explain two resource types, and introduce basic graph theory.

1.3.1 Basic Definitions in Deadlock Realm

Definitions of deadlock, livelock and deadlock avoidance in our context can be stated as follows.

Definition 1 *A system has a deadlock if and only if the system has a set of processes, each of which is blocked (e.g., preempted), waiting for requirements that can never be satisfied.*

Definition 2 *Livelock is a situation where a request for a resource is repeatedly denied and possibly never accepted because of the unavailability of the resource, resulting in a stalled process, while the resource is made available for other process(es) which make progress.*

Definition 3 *Deadlock Avoidance is a way of dealing with deadlock where resource usage is dynamically controlled not to reach deadlock (i.e., on the fly, resource usage is controlled to ensure that there can never be deadlock) [11, 16].*

Example 1 Deadlock

In an MPSoC application, on-chip processors may have to use several resources, for example, to process streaming data. Figure 1 shows such a system having two processors, a Very-Long Instruction Word (VLIW) Processor (VP) and a Specialized Processor (SP), and two resources, a Bluetooth Interface (BI) [56] and a Moving Picture Experts Group (MPEG) [55] decoder. Each processor (VP or SP) has to use both resources exclusively to complete its processing of the streaming data. In the case shown in Figure 1(b), VP holds resource MPEG while SP holds resource BI. (Please see the event sequence marked on the side of each edge shown in Figure 1(b).) Furthermore, VP requests BI, and SP requests MPEG. When SP requests MPEG, the system will have a deadlock since neither VP nor SP gives up or releases the resources they currently hold; instead, they wait for their requests to be fulfilled. ■

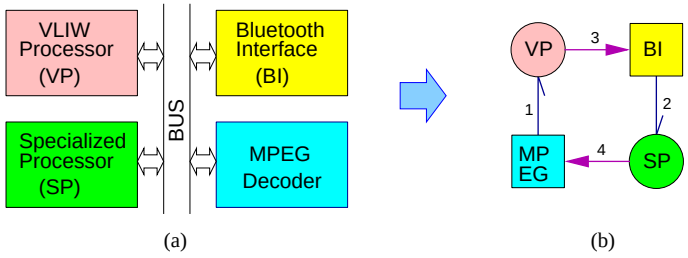


Figure 1: Deadlock example.

While all solutions presented in this thesis are applicable to single-instance resource systems, only one solution (the PBAU introduced in Chapter 5) can be used for multiple-instance resource systems; in order to make this distinction clear, we now define *single-instance resource* and *multiple-instance resource*.

Definition 4 *A single-instance resource is a resource that services no more than one process at a time. That is, while the resource is processing a request from a process, all other processes requesting to use the resource must wait [31].*

Definition 5 *A multiple-instance resource is a resource that can service two or more processes at the same time, providing the same or similar functionality to all serviced processes [31].*

Example 2 An example of a multiple-instance resource

A group of input/output (IO) buffers (e.g., ten IO buffers) can be considered as a multiple-instance resource. Rather than having each process keep track of each IO buffer, any request for an IO buffer is made to the group of IO buffers. In this way, not only can the overhead of tracking IO buffers for each process be reduced, but also interfaces between processes and IO buffers can be simplified because processes request from one place. ■

Example 3 Another example of a multiple-instance resource

The SoC Dynamic Memory Management Unit (SoCDMMU) dynamically allocates and deallocates segment(s) of global level two (L2) memory between PEs with very fast and deterministic time (i.e., four clock cycles) [46]. In a system having an SoCDMMU and 16 segments of global L2 memory, which can be considered as a 16-instance resource, rather than having each PE (or process) keep track of each segment, PEs request segment(s) from the SoCDMMU (which keeps track of the L2 memory). In this way, not only can the overhead of tracking segments for each PE be reduced but also interfaces between PEs and segments can be simplified because PEs request segment(s) from one place (i.e., the SoCDMMU). ■

Please note that a DSP processor or a co-processor can be categorized as either a master or a resource, depending on usage in specific applications. However, in our target MPSoC model (see Section 1.4.1) and our experiments we consider any DSP processor to be a resource. For the cases where a co-processor is dynamically changing its role back and forth between a master and a resource, further research is necessary.

The following necessary deadlock conditions have been stated in some form or another in previous work [8, 20, 31, 49]; nonetheless, for clarity, we state the following five conditions which, if present, indicate that a system has a deadlock.

Condition 1 *Mutual exclusion – resources cannot be shared.*

Please note that Condition 1 applies only for single-instance resource systems. A multiple-instance resource can be shared by no more than a certain number (i.e., the number of instances) of processes.

Condition 2 *No preemption – a resource can only be released by the process holding it.*

Condition 3 *Partial allocation – a process holding resource(s) can request additional resources.*

Condition 4 *Resource waiting – a process must wait for all requested unavailable resources to become available before proceeding.*

Condition 5 *Circular hold and wait – a closed chain of alternate processes and resources exists such that each process holds at least one resource needed by the next process in the chain, and no process can proceed without receiving all its requested resource(s).*

If all five of these conditions hold, then such a system has a deadlock. Please note that some conditions need not be strictly true at all times, but instead must be true only for the time(s) at which the system is deadlocked. For example, most modern chips have a "reset" pin that resets the entire chip, causing all the processes running on the chip's computational circuitry to release all held resources. Obviously, utilizing such a reset pin clears any current deadlocks by breaking Condition 2.

In addition, we further differentiate two kinds of deadlock: request deadlock (R-dl) and grant deadlock (G-dl). This distinction will become important in the deadlock avoidance algorithms we will propose later in this thesis.

Definition 6 *For a given system, if a request from a process directly causes the system to have a deadlock, then we denote this case as* ***request deadlock*** *or* ***R-dl****.*

In Example 1, the last request (i.e, SP requesting MPEG) causes a deadlock. We denote this case, in which a request causes a deadlock, as **request deadlock** or **R-dl**.

Definition 7 *For a given system, if the grant of a resource to a process directly causes the system to have a deadlock, then we denote this case as* ***grant deadlock*** *or* ***G-dl****.*

Example 4 Grant deadlock (G-dl)

We show a sequence of requests and grants that leads to a deadlock as shown in Figure 2. It is assumed that p_2 has a priority higher than p_3. At time t_1, process p_1 requests both q_1 and q_2, which are then granted to p_1. After that, p_1 starts working. At time t_2, p_3 requests q_2 and q_3. However, only q_3 is granted to p_3 since q_2 is unavailable. At time t_3, p_2 also requests q_2 and q_3, which are not available for p_2 yet. When the computation of p_1 is done, q_1 and q_2 are released by p_1 at time t_4 as shown in Figure 2(b). Then q_2 is granted to p_2 at time t_5 as shown in Figure 2(c) since p_2 has a priority higher than p_3. This last grant will lead to a deadlock in the system, which we denote as **grant deadlock** or **G-dl**. ■

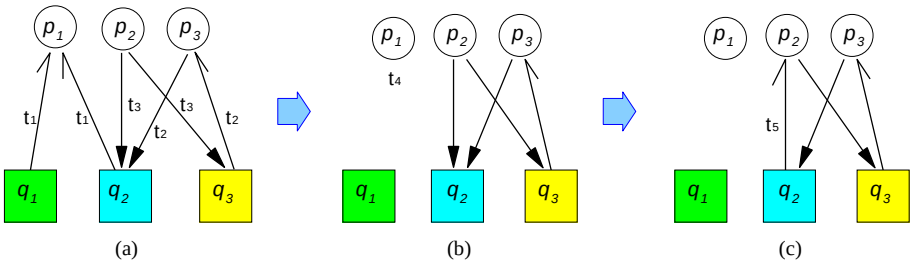


Figure 2: Grant deadlock (G-dl) example.

Please note that we differentiate between R-dl and G-dl because our deadlock avoidance algorithm in Chapter 4 requires this distinction to be made. The distinction is required because some actions can only be taken for either R-dl or G-dl; e.g., for grant deadlock (G-dl) it turns out that deadlock may be avoided by granting the resource to a lower priority process (see Example 4 where a final step of granting q_2 to p_3 – instead of to p_2 – could have avoided the deadlock).

We also introduce the definitions of an *H-safe sequence* and an *H-safe state* used to clarify the Parallel Banker's Algorithm in Chapter 5. Please note that the notion of "safe" was first introduced by Dijkstra [11] and was later formalized into "safe sequence," "safe state" and "unsafe state" by Habermann [16]. However, it turns out that a so-called "unsafe state" may in fact terminate normally (i.e., without deadlock); there do exist "unsafe

states" for which there exist sequences such that all processes terminate normally without any deadlock. For instance, in a system in an "unsafe state" (i.e., "unsafe" according to Habermann), if all processes voluntarily release resources they hold (i.e., not requesting up to their maximums), there will not be any deadlock. Thus, the implication that only a "safe state" executing a "safe sequence" can avoid deadlock is not true. As a result, we will refer to Habermann's "safe sequence" as an "H-safe sequence," to Habermann's "safe state" as an "H-safe state" and to Habermann's "unsafe state" as an "H-unsafe state" where the "H" stands for Habermann.

Definition 8 *An H-safe sequence is an enumeration $p_1, p_2, \ldots, p_n$ of all the processes in the system, such that for each $i = 1, 2, \ldots, n$, the resources that p_i may request are a subset of the union of resources that are currently available and resources currently held by $p_1, p_2, \ldots, p_{i-1}$ [11, 16].*

Please note the following.

(i) Any H-safe sequence can be proven to never evolve into deadlock.

(ii) We assume that the sequence is followed strictly (e.g., p_1 does not preempt p_3 in the middle of the sequence).

(iii) There are no promises about any timing properties such as periods and worst-case execution time.

(iv) We assume that processes having already finished do not execute again until the completion of all later remaining processes in the sequence.

Theorem 1 *A system of processes and resources is in an H-safe state if and only if there exists an H-safe sequence $\{p_1, p_2, \ldots, p_n\}$. If there is no H-safe sequence, the system is in an H-unsafe state [16].*

If a system is in an H-safe state, completion of all the processes can be guaranteed by restricting resource usage in the system with a strategy – such as the Banker's Algorithm [11, 16] – which executes one of the H-safe sequences. How a system in an H-safe

state cannot be in deadlock is shown in the following example.

Example 5 A strategy enforcing an H-safe sequence

Consider a system in an H-safe state with an H-safe sequence p_1, p_2, ..., p_n. Since the H-safe sequence starts with p_1, let p_1 finish by allowing only p_1 to allocate additional resources (i.e., currently available resources) (every other process requesting a resource must wait). Then, let p_2 finish by allowing only p_2 to allocate additional resources (i.e., currently available resources plus the resources that p_1 has released) (every other process including p_1 requesting a resource must wait). After that, let p_3 finish by allowing only p_3 to allocate additional resources (i.e., currently available resources plus the resources that p_1 and p_2 have released) (every other process requesting a resource must wait). In a similar fashion, let all remaining processes (i.e., $p_4, p_5, \ldots, p_n$) finish until every process has finished. Please note that as stated in the paragraph prior to Theorem 1 we assume that processes having already finished do not execute again until the completion of all later remaining processes in the sequence. ■

So far we showed fundamental definitions in the deadlock realm with some associated examples. In the next section, we will define terms used in graph theory on which our proofs in Chapters 3 and 4 rely.

1.3.2 Basic Definitions in Graph Theory

Here we introduce a few terms and definitions from graph theory, and examine how to represent the deadlock problem with a Resource Allocation Graph (RAG).

Definition 9 *Let $P = \{p_1, p_2, \ldots, p_n\}$ be a set of n requesters or processes that may request and/or hold a number of resources at any given time.*

Definition 10 *Let $Q = \{q_1, q_2, \ldots, q_m\}$ be a set of m resources that provide specific functions usable by processes.*

Definition 11 *Let the set of nodes V be $\{P \cup Q\}$, which is divided into two disjoint subsets P and Q such that $P \cap Q = \emptyset$. We also use another notation for the set of nodes, $V = \{v_1, v_2, \ldots, v_l\}$.*

Definition 12 *Let G be a set of grant edges. Let an ordered pair (q_i, p_j) be a grant edge where the first node is a resource $q_i \in Q$, the second node is a process $p_j \in P$ and q_i has been granted to p_j. Thus, a set of grant edges G can be written as $G = \big\{(q_i, p_j) \mid i \in \{1, 2, 3, \ldots, m\}, j \in \{1, 2, 3, \ldots, n\}$, and resource q_i has been granted to process $p_j\big\}$. An ordered pair (q_i, p_j) can also be represented by $q_i \rightharpoonup p_j$ or simply $g_{i \rightharpoonup j}$, where the harpoon "$\rightharpoonup$" represents a grant edge.*

Definition 13 *Let R be the set of request edges. Let an ordered pair (p_j, q_i) be a request edge where the first node is a process $p_j \in P$, the second node is a resource $q_i \in Q$, and p_j has requested q_i but has not yet acquired it. Thus, a set of request edges R can be written as $R = \big\{(p_j, q_i) \mid j \in \{1, 2, 3, \ldots, n\}, i \in \{1, 2, 3, \ldots, m\}$, and process p_j is requesting resource $q_i\big\}$. An ordered pair (p_j, q_i) can also be represented by $p_j \rightarrow q_i$ or simply $r_{j \rightarrow i}$, where the arrow "$\rightarrow$" represents a request edge.*

Definition 14 *Let edge set E be $\{R \cup G\}$. We also use another notation, $E = \{e_1, e_2, \ldots, e_h\}$.*

Definition 15 *A given system with processes and resources can be abstracted by a Resource Allocation Graph (**RAG**). A RAG is a directed graph $\gamma = \{V, E\}$, such that V is a non-empty set of nodes defined in Definition 11, and E is a set of ordered pairs of edges defined in Definition 14.*

Please note that the edge set E of a particular RAG may be empty at a moment when processes neither have any outstanding requests nor hold any resources.

Definition 16 *Given RAG γ, function $E(\gamma)$ produces the set of edges E of RAG γ.*

Since set V has two disjoint subsets P and Q, a process can only request a resource (not another process), and similarly a resource can only be granted to a process (but not to another resource); thus, any RAG γ is a bipartite graph, a graph whose vertices can be

partitioned into two groups $P \cup Q$ $(P \cap Q = \phi)$, where all edges cross from one group to the other group.

Definition 17 *We denote $\gamma_i = \{V, E\}$ as a particular system. We also define $\gamma_{i1}, \gamma_{i2}, \gamma_{i3}, \ldots$ to be different instances or **states** of the given system γ_i (same set V). Please note that the edge set $E(\gamma_{ij})$ is different for each $j \in \{1, 2, 3, \ldots\}$, but the node set $V = \{P \cup Q\}$ is constant for a given system γ_i. The system γ_i changes from one state γ_{ij} to another state γ_{ik} when handling requests, grants and releases of resources.*

Example 6 RAG

Figure 3(a) shows a RAG in state γ_{ij} converted to state γ_{ik} shown in (b) when a pending request (p_3, q_2) is granted. Here state γ_{ik} consists of a set of processes, $P = \{p_1, p_2, \ldots, p_6\}$, a set of resources, $Q = \{q_1, q_2, \ldots, q_6\}$, a set of request edges $R = \{(p_1, q_4), (p_3, q_1), (p_4, q_6), (p_5, q_4), (p_6, q_5)\}$ or $\{r_{1\to4}, r_{3\to1}, r_{4\to6}, r_{5\to4}, r_{6\to5}\}$, and a set of grant edges $G = \{(q_1, p_5), (q_2, p_3), (q_3, p_5), (q_4, p_3), (q_5, p_1), (q_6, p_2)\}$ or $\{g_{1\to5}, g_{2\to3}, g_{3\to5}, g_{4\to3}, g_{5\to1}, g_{6\to2}\}$. $E(\gamma_{ik})$, defined in Definition 16, gives $\{R \cup G\}$. ■

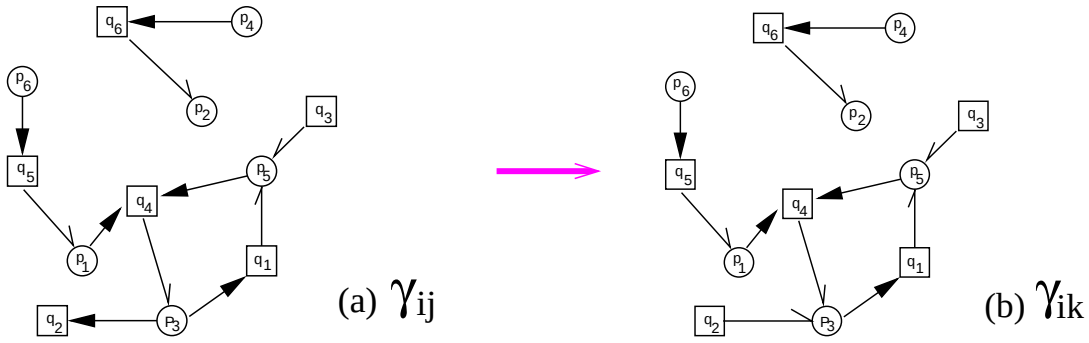


Figure 3: RAG example.

1.3.3 Definition of a Cycle

Both this subsection and the next subsection further refine various relationships among nodes and edges.

Definition 18 *A node v_τ is a **terminal node** iff the node v_τ has at least one edge and only incoming edge(s) or only outgoing edge(s). (A node with no edges is **not** a terminal node.)*

Definition 19 *An edge connected to a terminal node v_τ is called a* **terminal edge** *e_τ.*

Definition 20 *Given a RAG in state γ_{ij}, let $\tau(\gamma_{ij})$ be a function that returns the set $\{e_{\tau_1}, e_{\tau_2}, \ldots, e_{\tau_p}\}$ of all terminal edges in γ_{ij}.*

Definition 21 *A **link** node v_λ is a node that has exactly one incoming edge and exactly one outgoing edge. Clearly, the number of edges of a link node is two.*

Definition 22 *A **branch** node v_ω has one or more incoming edges and one or more outgoing edges such that the total number of edges is greater than or equal to three.*

Please note that while a resource node may have multiple incoming edges (multiple requests for the resource), it may have only one outgoing edge (the resource may be granted only to one process). A process node, in contrast, may have multiple incoming and outgoing edges.

Definition 23 *A node v_ϕ is a **connect** node if and only if node v_ϕ is either a link node or a branch node.*

Definition 24 *A **path** $(v_1, v_2, v_3, \ldots, v_{k-1}, v_k)$, $k \geq 2$ is a set of nodes connected by a consecutive ordered sequence of alternating request and grant edges (v_1, v_2), (v_2, v_3), $\ldots$, (v_{k-1}, v_k), where every node in the path is distinct and where every other node belongs to the same set P or Q.*

Please note that a path may have just one edge as well as many edges.

Definition 25 *A **simple path** is a path $(v_i, v_{i+1}, \ldots, v_k)$ such that both (i) v_i and v_k are terminal nodes and (ii) all other nodes v_j, $i < j < k$, are link nodes.*

Definition 26 *A **dangling path** is a path $(v_i, v_{i+1}, \ldots, v_j)$ such that either (i) v_i is a terminal node and v_j is a branch node or (ii) v_i is a branch node and v_j is a terminal node.*

Definition 27 *A **cycle** is a set of nodes* $(v_i, v_{i+1}, \ldots, v_j, v_i)$ *consisting of a path* $(v_i, v_{i+1}, \ldots, v_j)$ *and an additional edge between* v_j *and* v_i.

Please note that any subset $(v_k, v_{k+1}, \ldots, v_l) \subset$ cycle C with $\mid (v_k, v_{k+1}, \ldots, v_l) \mid \geq 2$ is a path. Please note also that all nodes in a cycle are connect nodes.

Example 7 Cycle

The RAG in Figure 4 contains terminal nodes and edges; link, branch and connect nodes; simple and dangling paths; and a cycle. $\{p_2, p_4, p_6, q_2, q_3\}$ are terminal nodes with corresponding terminal edges $\{(q_6, p_2), (p_4, q_6), (p_6, q_5), (q_2, p_3), (q_3, p_5)\}$, respectively. Link nodes are $\{q_6, q_5, p_1, q_1\}$, branch nodes are $\{q_4, p_3, p_5\}$, and connect nodes are all the link and branch nodes, i.e., $\{q_6, q_5, p_1, q_1, q_4, p_3, p_5\}$. An example of a simple path is path $p4 \rightarrow q6 \rightharpoonup p2$. An example of a dangling path is path $p6 \rightarrow q5 \rightharpoonup p1 \rightarrow q4$. Finally, $p3 \rightarrow q1 \rightharpoonup p5 \rightarrow q4 \rightharpoonup p3$ forms a cycle. Please note that while the formal notation for the cycle is $(p_3, q_1, p_5, q_4, p_3)$, we also use $p3 \rightarrow q1 \rightharpoonup p5 \rightarrow q4 \rightharpoonup p3$ as an informal notation for the same cycle. ■

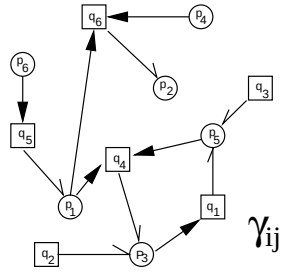


Figure 4: Cycle example.

Definition 28 *The **size of a cycle** is the number of nodes (both resources and processes) involved in the cycle.*

1.3.4 Definition of a Terminal Reduction Step

Having introduced a RAG and a cycle, we now define a terminal reduction step and its related properties used to reveal a deadlock quickly and efficiently.

Definition 29 *A **terminal reduction step** is a step in which at least one terminal edge* e_τ *is removed from the RAG under consideration.*

Definition 30 *The application of a terminal reduction step to γ_{ij}, resulting in a distinct state $\gamma_{i,j+1}$, is called the **reduction** of γ_{ij} to $\gamma_{i,j+1}$. Furthermore, we also say that state γ_{ij} has been **reduced** to $\gamma_{i,j+1}$.*

Definition 31 *If a system state γ_{ij} can be transformed by a terminal reduction step to another state $\gamma_{i,j+1}$, resulting in $\gamma_{i,j+1} \neq \gamma_{ij}$, then the system state γ_{ij} is said to be **reducible**. If a system state γ_{ij} cannot be reduced to another different state $\gamma_{i,j+1}$ (because there are no terminal edges to which to apply a terminal reduction step), then system state γ_{ij} is said to be **irreducible**.*

Definition 32 *A system state $\gamma_{i,j+k}$ is said to be **completely reduced** if $E(\gamma_{i,j+k}) = \emptyset$. Otherwise, a system state $\gamma_{i,j+k}$ is said to be **incompletely reduced** if $E(\gamma_{i,j+k}) \neq \emptyset$.*

1.4 Target System

1.4.1 Target Multiprocessor System-on-a-Chip

To illustrate our target system, let us show an MPSoC example.

Example 8 A future Request-Grant MPSoC

We introduce the device shown in Figure 5 as a particular MPSoC example. This MPSoC consists of four Processing Elements (PEs) and four resources – a Video and Image capturing Interface (VI), an MPEG encoder/decoder, a DSP and a Wireless Interface (WI), which we refer to as q_1, q_2, q_3 and q_4, respectively, as shown in Figure 5(b). The MPSoC also contains memory, a memory controller, a DDU and a DAU. In the figure, we assume that each PE has only one active process; i.e., each process p_1, p_2, p_3 and p_4, shown in Figure 5(b), runs on PE1, PE2, PE3 and PE4, respectively. In the current state, resource q_1 is granted to process p_1, which in turn requests q_2. In the meantime, q_2 is granted to p_3, which requests q_4, while q_4 is granted to process p_4; the resulting system state is shown in Figure 5(b). The DAU in Figure 5 receives all requests and releases, uses the DDU to decide whether or not a particular request or grant can cause a deadlock and then permits the request or grant only if no deadlock results. ■

Figure 5 shows our primary target MPSoC consisting of multiple processing elements with L1 caches, a large L2 memory, and multiple hardware IP components with essential

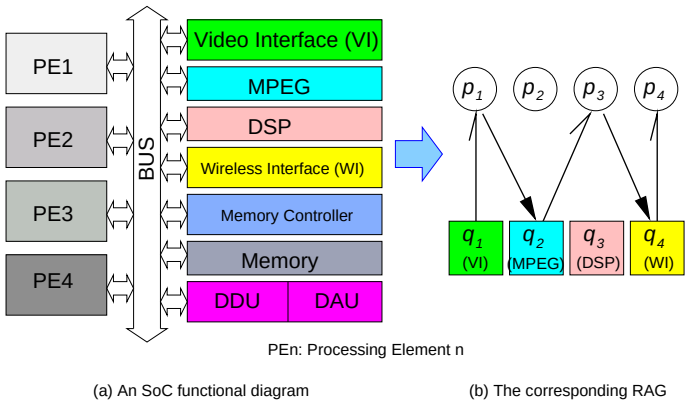


Figure 5: Practical MPSoC realization.

interfaces such as a memory controller, an arbiter and a bus system. We consider this kind of request-grant system as our system model in the view of deadlock. For each specific deadlock solution, there may be minor variations from this model. Based on our system model, we next introduce a Real-Time Operating System (RTOS) we use for system simulation and then mention some underlying assumptions related to our deadlock research in such MPSoCs.

1.4.2 Target Real-Time Operating System

To benefit from an RTOS, we use Atalanta RTOS version 0.3 [51], a small and configurable shared-memory multiprocessor RTOS developed at the Georgia Institute of Technology. Code of Atalanta RTOS version 0.3 resides in shared memory, and all PEs execute the same RTOS code and share kernel structures as well as the states of all processes and resources; currently Atalanta RTOS version 0.3 supports only use of all PowerPC processors or only all ARM processors. Atalanta supports priority scheduling with priority inheritance as well as round-robin; task management such as task creation, suspension and resumption; various Inter Process Communication (IPC) primitives such as semaphores, mutexes, mailboxes, queues and events; memory management; and interrupts. Please note that any other operating system can be targeted (instead of using Atalanta) with associated straightforward engineering effort.

1.5 Assumptions

Considering this kind of future MPSoC shown in Figure 5(a) as our system model, we now introduce some of our assumptions about future MPSoC designs related to analyzing deadlock in such MPSoCs.

Assumption 1 *In our system model, only reusable resources exist.*

A **reusable resource** is characterized as follows: (i) units are neither created nor destroyed (fixed total inventory), and (ii) units are requested and acquired by processes from a pool of available units. When a process finishes using an acquired reusable resource, the resource is returned to the resource pool so that other processes can have a chance to use the resource. In this thesis, all further references to "resource" should be read as "reusable resource."

Assumption 2 *In our system model, there exists a fixed number of resources.*

Please note that the following assumption (Assumption 3) is *not* applied to PBAU but is only applied to the DDU and DAU (note that the concept of the PBAU, DDU and DAU were introduced in Section 1.2 and will be explained in great detail in Chapters 3, 4 and 5).

Assumption 3 *Each resource has one unit. Furthermore, each resource can serve only one process at any given time. As a result, a process must wait for all required unavailable resources to become available before proceeding.*

Assumption 4 *A resource can be released only by the process holding it.*

Please note that in some situations, Assumption 4 may not hold. For example, if the system is reset, then all resources will be released, and all processes will be restarted. However, our analysis is intended to address the normal operation of an SoC. During normal operation, we assume that once a resource is granted to a process, only the owner process can release the resource. In the case that system reset occurs, any deadlock detection operation in progress will have to be restarted with the new system state after reset.

Assumption 5 *Resources are preemptible.*

Assumption 6 *The RTOS or other software provides a mechanism that can ask a process to release any resource(s) the process currently holds.*

Assumption 7 *While a process holds some resources, the process can request additional resources.*

Assumption 8 *All requests and releases in the system are serialized by some kind of mechanism such as bus arbitration among multiprocessors. Thus, at any instant, there exists only one outstanding activity of a request or release.*

Please note that a request or release could involve multiple resources as well as multiple instances. Handling multiple resources will either be serialized inside the DDU or DAU, or be successfully processed inside PBAU. Please note also that if there are multiple requests and/or multiple releases from multiple processes at the same time, the requests and/or releases must be serialized one by one by some kind of mechanism such as a queue and/or bus arbitration and then fed to deadlock solutions before being processed.

1.6 Thesis Organization

The thesis is organized into seven chapters:

CHAPTER I: INTRODUCTION. This chapter provides a general overview of deadlock related problems. The chapter also provides some terminology used in the deadlock realm. Finally, this chapter summarizes the contributions of this thesis.

CHAPTER II: MOTIVATION AND PREVIOUS WORK. This chapter first addresses our motivation for this research, then describes previous work in deadlock research, and lastly shows notable differences between our solutions and previous work.

CHAPTER III: PROOFS OF THE CORRECTNESS AND RUN-TIME COMPLEXITY OF THE DDU. This chapter first introduces Parallel Deadlock Detection Algorithm (PDDA) and then proves the correctness and run-time complexity of the DDU. After that, the chapter describes the DDU architecture and presents synthesis result. Lastly, Chapter III shows algorithm run-time as well as application execution time comparisons among three deadlock detection algorithms (one of which is the DDU).

CHAPTER IV: DEADLOCK AVOIDANCE UNIT. This chapter introduces a novel deadlock avoidance algorithm and presents a hardware implementation of the algorithm. This chapter also shows execution time comparison as well as application run-time comparison between the deadlock avoidance algorithm and its hardware implementation.

CHAPTER V: PARALLEL BANKER'S ALGORITHM UNIT. This chapter describes a novel Parallel Banker's Algorithm (PBA) and its hardware implementation, which we call PBA Unit (PBAU), and shows algorithm run-time as well as application execution time comparisons between PBAU and the Banker's Algorithm in software.

CHAPTER VI: INTEGRATION INTO THE δ HW/SW RTOS PARTITIONING FRAMEWORK. This chapter expresses the integration of deadlock hardware solutions into the δ hardware/software RTOS partitioning framework that has been used to configure and generate simulatable RTOS/MPSoC designs. This chapter also describes an IP generation tool used to automatically generate a hardware deadlock solution out of the DDU, DAU and PBAU according to the numbers of processes and resources that a user specifies.

CHAPTER VII: CONCLUSION. This chapter summarizes the major accomplishments of this thesis.

CHAPTER II

MOTIVATION AND PREVIOUS WORK

2.1 Motivation

Recent technology trends show that System-on-a-Chip (SoC) technology enables a multi-core multithreaded system on a single chip. An example of such an SoC is the Xilinx Vertex II Pro [57], which may contain multiple PowerPC processors and additional Intellectual Property (IP) cores. Furthermore, due to the ever increasing expansion of the Internet and wireless communication, a tremendous amount of multimedia related data is being created, modified and exchanged; this multimedia data is becoming larger with more varied and complicated encodings, requiring unprecedented processing power. To support such multimedia communication, numerous algorithms, specialized processors, image/video coding hardware modules and error detection/correction modules have been implemented and exploited [12]. Given these trends, we predict that in the near future, MPSoC designs will have many Processing Elements (PEs) and hardware resources, which is the way MPSoC will rapidly evolve.

Therefore, we predict that, in future MPSoCs, many processes will concurrently run and dynamically require and access such available on-chip resources. Accordingly, systems will handle much more functionality, enabling much higher levels of concurrency and requiring many more deadlines to be satisfied. Not only that, but ensuring predictability and reliability in such MPSoCs will be much more difficult. As a result, we predict there will be resource sharing problems among the many processors desiring the resources, which may result in some kind of deadlock more often than designers might realize.

In most current embedded systems, deadlock is not a critical issue due to the use of only a few (e.g., two) processors and a couple of custom hardware resources (e.g., direct

memory access hardware plus a video decoder). However, in the coming years future chips may have five to twenty processors and ten to a hundred resources all in a single chip as shown in Figure 6. In such systems, we predict that deadlock possibilities will no longer be ignorable issues, but will, if not properly addressed, become problems in the sharing of resources.

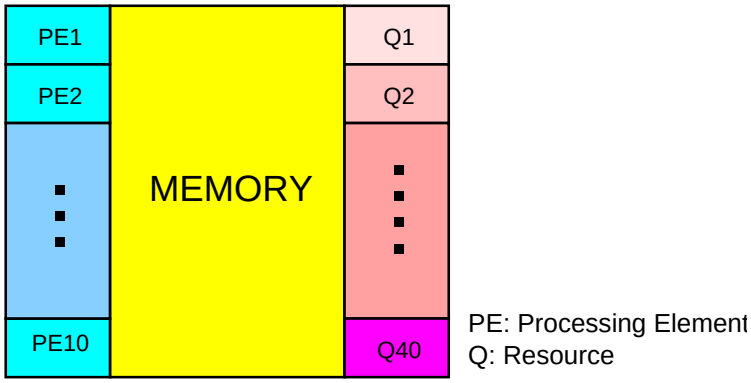


Figure 6: Future MPSoC.

How can we efficiently and timely cope with deadlock problems in such MPSoCs? We envision that although MPSoC may produce deadlock problems, an MPSoC architecture can also provide efficient hardware solutions to deadlock. Thus, this thesis describes such solutions, i.e., operation and proofs of Parallel Deadlock Detection Algorithm (PDDA); a novel Deadlock Avoidance Algorithm (DAA) and its hardware implementation, the Deadlock Avoidance Unit (DAU); and a hardware implementation of a novel Parallel Banker's Algorithm. The solutions presented in this thesis can improve the reliability and correctness of applications running on an MPSoC under a Real-Time Operating System (RTOS). Of course, adding a centralized module on an MPSoC may lead to a bottleneck. However, since resource allocation and deallocation are preferably managed by an operating system (which already implies some level of centralized operation), adding hardware can potentially reduce the burden on software rather than becoming a bottleneck.

2.2 Software Deadlock Research

2.2.1 Overview of Prior Deadlock Research

Researchers have put tremendous efforts into deadlock research, three well-known areas of which are deadlock detection, prevention and avoidance. Among them, *deadlock detection* does not limit a system's freedom in any way since deadlock detection does not typically restrict the behavior of a system, facilitating full concurrency. Deadlock detection, however, usually requires a recovery once a deadlock is detected. In contrast, *deadlock prevention* prevents a system from reaching deadlock typically by constraining request orders to resources in advance, resulting in the fact that deadlock never occurs (i.e., deadlock prevention is extremely conservative). However, any such strict constraint on requests may limit concurrency and thus degrade performance. One benefit though is that prevention may not require invocation of a prevention algorithm on every event of a request or a release; that is, prevention strategies can be devised which are correct-by-construction (i.e., are guaranteed to work due to the restrictions in place but without requiring additional "checking" or any other code to run dynamically). By contrast, as stated in Definition 3, *deadlock avoidance* is accomplished by dynamically controlling resource usage (i.e., allowing or denying requests or grants) whenever a request or a release event occurs. In deadlock avoidance, resource accesses are allowed as long as the system remains in a safe state (i.e., not resulting in deadlock – please note that H-safe states are a subset of safe states, where we define a safe state to be a state which has an execution sequence not resulting in deadlock). In other words, deadlock prevention is done statically while deadlock avoidance is done on the fly. Thus, it is well known that deadlock avoidance typically involves less restrictions and results in higher resource utilization than deadlock prevention [50].

Figure 7 represents our view of the relationship between deadlock avoidance and deadlock prevention. The distinction is primarily made based on whether deadlock is done statically prevented in advance or deadlock is dynamically avoided on the fly. Please note that our view of separating deadlock avoidance and prevention may be different from

others' point of view. For instance, Habermann called his Banker's Algorithm a prevention method [16]. However, our classification agrees to the viewpoint of most authors [8, 17, 31, 50].

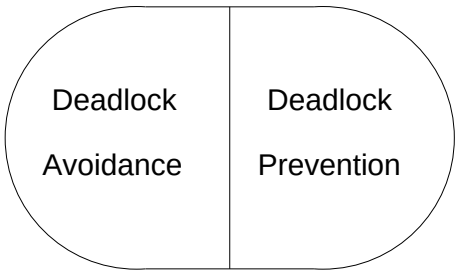


Figure 7: Relationship between deadlock avoidance and deadlock prevention.

2.2.2 Deadlock Detection

All software deadlock detection algorithms to date have a run-time complexity of at least $O(m \times n)$ or $O(m^3)$, where m is the number of resources and n is the number of processes (please note that the number of edges in a RAG is $O(m \times n)$). In 1970, Shoshani et al. proposed an $O(m \times n^2)$ run-time complexity detection algorithm [49], and about two years later, Holt proposed an $O(m \times n)$ algorithm to detect a knot that tells whether or not deadlock exists [20]. Holt's model of multiple processes and resources provides a versatile representation of resource allocation, and the approach describes a general resource system that models consumable as well as reusable resource types. Both of the aforementioned algorithms (of Shoshani et al. and of Holt) are based on a Resource Allocation Graph (RAG) representation. Leibfried proposed a method of describing a system state using an adjacency matrix representation and a corresponding scheme that detects deadlock with matrix multiplications but with a run-time complexity of $O(m^3)$ [29].

In traditional deadlock detection algorithms, each time a request, a grant, or a release event occurs, the event is reflected on a RAG, and then a search is carried out for a cycle, which, if found, indicates that the system corresponding to the RAG has a deadlock. Kim and Koh proposed a new deadlock detection method [22]. The approach of Kim and Koh

is somewhat different from traditional ones in that their method considers each separate subgraph in a RAG as a tree. Hence, each tree has a root node (which corresponds to an active process), and a RAG may have many trees. The proposed method detects a deadlock as soon as a root node requests a resource already belonging to the same tree that the root node belongs to. This can be implemented by associating each resource with the identifier of a tree to which the resource belongs. Since the authors' method is based on constructing trees of a sequence of request and grant events, when multiple requests and grants occur at a particular instant in the system under consideration, the overhead must be accounted for. That is, Kim and Koh's approach has $O(m \times n)$ run-time for "detection preparation"; thus an overall run-time for detecting deadlock (starting from a system description that just came into existence, e.g., due to multiple grants and requests occurring within a particular time or clock cycle) of at least $O(m \times n)$ [22].[1] A disadvantage of the Kim and Koh's method is that, as the authors admit, the worst case execution time of their release algorithm takes $O(m + n)$, as opposed to $O(1)$ in traditional deadlock detection algorithms.

In deadlock detection, however, once deadlock is detected, there must be some way of breaking out of the deadlock, which is called "recovery from deadlock" or "deadlock resolution." Typical deadlock resolution methods include resetting the system, aborting process(es), rolling back to a state before deadlock, and releasing resource(s). We do not address these methods further because the solution of deadlock recovery is not within the scope of this research.

2.2.3 Deadlock Prevention

Deadlock can be prevented by designing a system such that one of the deadlock conditions (i.e., Conditions 1–5 in Section 1.3) can never occur; thus, deadlock would be impossible,

[1] Please note that Kim and Koh claim that their deadlock detection algorithm could be performed in $O(1)$ run-time. However, the $O(1)$ run-time can only be achieved if the overhead of detection preparation time can be ignored. Thus, the run-time complexity (i.e., the worst-case) of their algorithm is $O(m \times n)$.

hence the name “deadlock prevention” which seems very attractive [8]. In general, however, designing a system such that one of the deadlock conditions is guaranteed to never occur will degrade system performance significantly, which we address here.

Since there are five deadlock conditions, breaking these conditions could suggest five approaches to deadlock prevention. However, due to intrinsic attributes of resources, some conditions are inevitable. Let us first consider Condition 1, mutual exclusion. There might be some resources that can be shared such as read-only files or programs. However, it is typical that resources cannot be shared; that is, Condition 1 is typically unavoidable. In the consideration of Condition 4, there can be no way to break this condition unless a process can proceed without a required resource, which is almost never the case. Therefore, there can be three possible prevention methods remaining.

One method able to break Condition 3 is the collective-request method, in which a process always makes requests of all its required resources at the same time or is blocked until all requests can be granted together, meaning that no incremental request is allowed. Therefore, a process must request all the resources that it will ever require during its lifetime in the beginning of its execution, or whenever a process requires additional resource(s), the process must first release all the resources that it currently holds and then issue a new request that includes all the resources it currently needs, thereby avoiding Condition 3, partial allocation. However, this method inevitably causes resource underutilization and/or process starvation in most practical situations.

Another method able to break Condition 5 is the ordered-request method, in which all resources are numbered in a specific order such as a priority order. Thus, all processes request resources in that order. For example, all resources are assigned with distinct prioritized numbers. Then, a process can only request a resource with a number (priority) greater than the number (priority) of any resource that the process currently holds. This method prevents a system from forming a cycle in the resource graph and thus keeps a system from having a deadlock at all by permanently avoiding any circular waiting, i.e.,

Condition 5. However, similar to the collective-request method, this method may result in poor utilization of resources due to the restriction of a resource request order.

Another somewhat forcible way of deadlock prevention would be the resource preemption method, breaking Condition 2. However, the method of breaking Condition 2 is typically categorized not into deadlock prevention but into deadlock avoidance or deadlock recovery. Thus, we describe this method in the next subsection.

2.2.4 Deadlock Avoidance

A traditional well-known deadlock avoidance algorithm is the Banker's Algorithm (BA) [11]. BA requires each process to declare the maximum requirement (claim) of each resource the process will ever need. Then, while requests and releases are being made, the algorithm allows requests only if the system remains in an H-safe state, resulting in that even in the worst-case where all processes request their maximum claims, any sequence of process executions allowed by BA results in all requests eventually being fulfilled. Consequently, even though some resources may be available for a particular request, due to the possibility of other processes potentially requesting their maximum claims, some requests are denied which in fact could have been fulfilled without resulting in deadlock.

In 1999, Lang proposed a variant of BA with an $O(m \times n)$ run-time complexity [23]. Lang's approach decomposes trees of a Resource Allocation Graph (RAG) into regions and computes the associated maximum claims, prior to process execution (note that Lang's tree is a subgraph of a RAG where the root of the subgraph is a process which currently holds all necessary resources). By more accurately calculating an optimal set of maximum claim estimates in each region, Lang's algorithm may improve resource utilization as compared to BA that uses global maximum claims.

However, the requirement of advance knowledge about the maximum necessary resource usage for all processes in a system in BA as well as its variants unfortunately makes the implementation of such a method difficult in real systems with dynamic workloads. In

general, although BA and its variants guarantee to the avoidance of deadlock, they may be impractical in many systems because of the following disadvantages: (i) the avoidance algorithm must be executed for every request prior to granting a resource; (ii) the deadlock avoidance algorithm restricts granting of requests leading to an H-unsafe state, which may reduce resource utilization (since an H-unsafe state may not necessarily lead to deadlock), degrading system performance; (iii) the maximum resource requirements (and thus requests) might not be known in advance (e.g., with a program with conditional execution); and (iv) the maximum number of processes must be known [8, 11].

However, since there are states (including some H-unsafe states) that may not evolve to deadlock, if the concept of an H-safe state can be relaxed to only satisfy our definition of deadlock avoidance (Definition 3), some of the disadvantages of deadlock avoidance can be lifted. For instance, if resource preemption is allowed, then one of processes involved in a potential deadlock can be asked to release the resource(s) involved in the potential deadlock to break such a potential deadlock (where the process has to wait and later rerequest the resource(s) it requires). This type of deadlock avoidance method may incur a high penalty, even requiring checkpointing to rollback in some cases. However, benefits of this are the following: (i) this method may improve resource utilization by relaxing the concept of an H-safe state and (ii) resource preemption is better than process preemption, which is a way to recover from deadlock. This preemption method of deadlock avoidance can also be categorized into the deadlock detection and recovery scope, mentioned earlier. Using this method may eliminate the requirement of maximum claim declaration.

These insights were taken by Belik. An approach utilizing benefit (i) in the above paragraph is Belik's method [7]. In 1990, Belik proposed a deadlock avoidance technique in which a path matrix representation is used to detect a potential deadlock before the actual resource allocation. However, Belik's method requires $O(m \times n)$ run-time for updating the path matrix in releasing or allocating a resource and thus an overall complexity for avoiding deadlock of $O(m \times n)$. Furthermore, Belik does not mention any solution to

livelock although livelock is a possible consequence of his deadlock avoidance algorithm.

Please note that in Section 2.4 we will compare approaches of this thesis with these prior software approaches.

2.3 Hardware Deadlock Research

Although there have been many innovative ideas and software algorithms introduced to effectively detect, avoid and/or prevent deadlock [7, 11, 14, 16, 17], for various reasons most of these approaches have not been exploited in practical systems. Primary reasons we conjecture are (i) the time-consuming software run-time and (ii) no necessity so far. In fact, the general deadlock problem has been shown to be NP-complete [15]. In other words, utilizing a software deadlock algorithm in an MPSoC may incur a fair amount of loss of computational power, which otherwise could have been used for useful work. Thus, a better way of overcoming the drawback of using a software deadlock algorithm for an MPSoC is to implement the deadlock algorithm in hardware so that while deadlock is efficiently, quickly and silently (i.e., unnoticed by the users or programmers) detected and/or avoided, the applications achieve their designated goals with almost no sacrifice in system performance. That is, by adding a small amount of hardware to the MPSoC, a deadlock solution will be able to become worth running since performance will hardly be affected at all and since deadlock will be detected and/or avoided.

2.3.1 Deadlock Detection

To realize such a possibility of hardware implementation of a deadlock detection algorithm, Parallel Deadlock Detection Algorithm (PDDA) and its hardware implementation in the Deadlock Detection Unit (DDU) have been proposed [48]. Figure 8 shows the architecture of the DDU for three processes and three resources. This architecture will be explained in great detail in Section 3.3.

The DDU takes $O(1)$ run-time for updating a state matrix in requesting, releasing or allocating a resource. Furthermore, the DDU has a complexity of $O(min(m, n))$ in detecting deadlock, which we prove in Section 3.2.5. Such low run-time is achieved by (i) utilizing hardware parallelism and (ii) using a simple two-bit binary representation of the types of each edge: the request edge of a process requesting a resource, the grant edge of a resource granted to a process, or no activity (neither a request nor a grant) [27]. PDDA distinguishes itself from others in that PDDA deals with the edges that are not involved in cycle(s), as opposed to other algorithms that try to find exact cycles, which typically requires more computational time. Furthermore, PDDA does not require linked lists. Not only that, but by implementing PDDA with a small amount of hardware, the designed deadlock detection unit hardly affects system performance (and potentially has no negative impact whatsoever) yet provides the basis for enhanced deadlock detection.

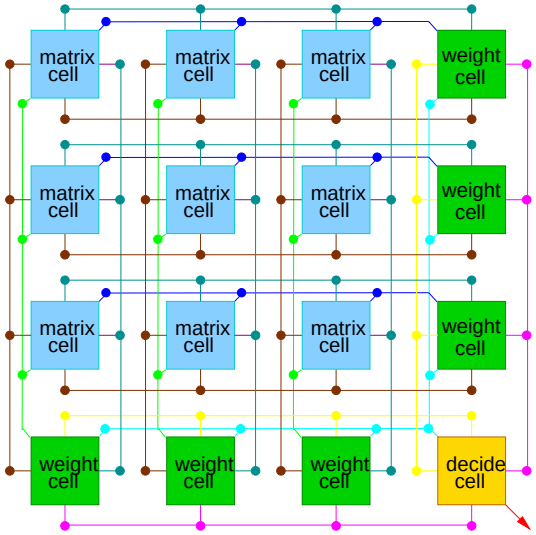


Figure 8: DDU architecture.

However, the previous authors neither implemented PDDA in software nor formally proved the correctness and run-time complexity of the proposed DDU, which we prove in Sections 3.2.4 and 3.2.5. Moreover, we demonstrate extensive comparisons among the DDU, PDDA in software and an $O(m \times n)$ deadlock detection algorithm in Section 3.4.1.

Not only that, we describe detailed explanation of PDDA in Section 3.2.3 as well as the circuitry of the DDU in Section 3.3.4.

2.3.2 Deadlock Avoidance

Although many deadlock avoidance approaches have been introduced so far [7, 8, 11, 13, 14], to the best of our knowledge, there has been no prior work in a hardware implementation of deadlock avoidance. Thus, this thesis plus associated publications by the author appear to be the first known work presenting deadlock avoidance in hardware.

2.4 Our Approaches Compared to Prior Work

On the contrary to previous deadlock research, our approach using the Deadlock Avoidance Unit (DAU) not only overcomes some disadvantages, such as (iii) and (iv) mentioned in the third paragraph of Section 2.2.4, but also resolves the livelock associated with deadlock avoidance, which we will explain in detail in Chapter 4. Please note that the DAU utilizes resource preemption to break such livelock. The DAU reduces the deadlock avoidance time by over 99% *(about 300X)* and achieves in a particular example approximately 40% speedup of application execution time as compared to the execution time of the same application using the same algorithm in software.

As opposed to BA and its variants, our Parallel Banker's Algorithm (PBA) presented in Chapter 5 implements in parallel Habermann's variant of the Banker's Algorithm so that PBA can achieve an $O(n)$ run-time complexity in a hardware implementation as compared to an $O(m \times n^2)$ complexity of Habermann's BA. In fact, the PBA Unit (a hardware implementation of PBA) achieves about a 1600X speedup of the average algorithm execution time and gives in a particular example a 19% speedup of application execution time over avoiding deadlock with BA in software.

We also considered deadlock prevention, but because prevention requires that each process conform to the prevention policy implemented in the system in advance, which would

impose overhead of necessary operation (e.g., keeping an order of requests) on each process (i.e., an application), we decided not to further investigate hardware approaches to deadlock prevention.

2.5 Summary

In this chapter, we provide motivation for deadlock hardware solutions by addressing recent technology trends. We further present some prior work in deadlock research and briefly mention the novelty of our research. In the next few chapters, we will further describe our approaches in detail. Specifically, in the next chapter, we describe our research regarding PDDA and the DDU.

CHAPTER III

PROOFS OF THE CORRECTNESS AND RUN-TIME COMPLEXITY OF THE DDU

3.1 Introduction

In this section, we will first show how the Deadlock Detection Unit (DDU), the hardware implementation of Parallel Deadlock Detection Algorithm (PDDA), can be used in a multiprocessor multiresource SoC such as the MPSoC shown in Figure 5(a). After that, we will introduce and prove our deadlock theorems and direct consequences under our assumptions described in Section 1.5 (i.e., our deadlock theorems are modified from general deadlock theorems [31] – specifically, the modifications accommodate our hardware-centric notation, thus easing generation of proofs about our hardware operation). Then, we will present a translation of a system state γ_{ij} from a RAG into a matrix. This matrix representation forms the basis of PDDA and enables the implementation of simple but very fast parallel deadlock detection in hardware. Using the matrix representation, we will next define terminal rows and terminal columns to which a novel parallel terminal reduction step (the core of PDDA) can be applied. After that, we will describe PDDA and then prove that the DDU has a run-time complexity of $O(min(m, n))$, where m and n are the numbers of resources and processes, respectively, involved in deadlock detection. Finally, we will describe detailed PDDA operation with mathematical representations and detailed DDU architecture and demonstrate extensive experimentation [27].

3.1.1 Deadlock Detection Unit (DDU) Operation in a System

First let us briefly explain how the DDU operates and how it can be used in a system. Please note that we assume that the maximum number processes as well as the maximum number

resources are fixed in advance. The DDU idles when there is no request or grant. That is, the DDU becomes active and starts working only when a request or grant event occurs. Once the DDU is activated, it operates in only a few clock cycles (at most $2 \times min(m, n) - 3$ cycles, as proven in Section 3.2.5) and then produces a deadlock detection result. After that, the DDU returns to an idle state and remains idle until another event occurs.

Please note that with PDDA implemented in the DDU hardware, the DDU decides whether a given system state has a deadlock or not based on the requests and grants that have occurred, not on any future events.

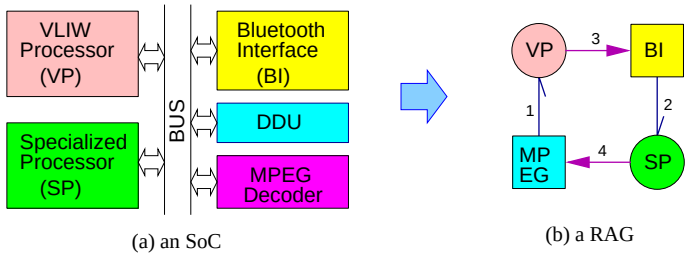
The DDU can be employed in such a way that while processes request resources randomly and directly from the resources (i.e., without the intervention of the DDU), the DDU just monitors these requests and grants. As soon as a deadlock is detected, however, the DDU notifies the RTOS or other application software of the existence of a deadlock, in which case the RTOS or other application software may release some resources or take other actions to break the deadlock. While detecting deadlock, the DDU emits a busy signal, indicating that the DDU is currently working, thus temporarily preventing any further request or grant events. Since the deadlock detection takes only a few clock cycles, any further requests or grants during this short amount of time can be temporarily queued in hardware; then, after the current detection, the next deadlock detection will start with the update of the queued events. In this way of DDU usage, the DDU does not impede normal system performance at all or any impact is minimal.

In addition to the one way of DDU usage described here, there may be more ways to utilize the DDU; we will discuss one such other way in the next chapter which focuses on deadlock avoidance. However, the focus on this chapter is to prove the correctness and run-time complexity of the DDU. The following example represents the DDU usage described in the previous paragraph.

Example 9 DDU usage

From Example 1, if a DDU is employed, the system would look like Figure 9. In this MPSoC, the

DDU monitors resource request and grant activities. In the case shown in Figure 9, we assume both VP and SP each receives a stream to be processed almost at the same time. While VP first requests and holds resource MPEG, SP requests and holds resource BI. (Please see the event sequence marked on the side of each edge shown in Figure 9(b).) In the meantime, the DDU executes PDDA, which checks for deadlock whenever a request or grant event occurs, but the DDU fails to find a deadlock so far. After that, VP requests and waits for BI, which has already been granted to the SP. Next, SP requests MPEG, and then the DDU starts to determine if a deadlock exists. Since the system state has a deadlock at this instant, the DDU will find the deadlock; thus, the DDU will indicate that a deadlock exists. ■



(a) an SoC (b) a RAG

Figure 9: DDU usage example.

Having introduced a way to use the DDU, in the next section we will prove the deadlock theorems and run-time complexity of the DDU under the assumptions of our target system described in Sections 1.4.1 and 1.5.

3.2 Proofs of the correctness and run-time complexity of the DDU

3.2.1 Preliminary Theorems

Before introducing and proving our deadlock theorems, we first describe one more definition and two underlying assumptions as they relate to the operations of the DDU in practical situations as well as to our deadlock theorems.

Definition 33 *A process is **making progress** in a system γ_i in state γ_{ij} only when one of the following is true: (i) when a process does not hold resources, the process, if given control of*

a processor (e.g., by a priority scheduler), could currently perform its computation without the need of any resources, or (ii) when a process holds some resources, the process, if given control of a processor, would currently use the resources to perform its computation without the need of any additional resources.

Assumption 9 *In a practical situation, at the instant when the DDU becomes active and is checking for deadlock, the DDU determines deadlock based on the requests and grants currently in existence at that instant,* ***not on any future events*** *(i.e., assuming no additional requests or grants are accepted when the DDU is in operation, which can easily be implemented by indicating a "busy" signal).*

Assumption 10 *At the instant when the DDU becomes active (at a particular state γ_{ij} of a system), if a process is making progress and using some resources, then it is assumed that the process has all the resources it requires and that it can and will finish using its resources within a finite time, thus eventually releasing all the resources that the process has used.*

We now introduce and prove our deadlock theorems and their direct consequences.

Corollary 1 *In a system state γ_{ij}, the number of nodes involved in the smallest possible cycle is four. Similarly, the number of edges involved in the smallest possible cycle is four.*

Proof: Since a RAG γ is a bipartite graph (by Definitions 11 and 15), γ_{ij} cannot have any edge from process p_g to process p_h for any two processes $p_g, p_h \in V(\gamma_{ij})$. Similarly, γ_{ij} cannot have any edge from resource q_k to resource q_l for any two resources $q_k, q_l \in V(\gamma_{ij})$. Since we need to find the minimum number of nodes that can form a cycle, let us consider case (i) where one process and one resource exist. Case (i) can have a path between the two nodes but cannot form a cycle because according to Assumption 3 there cannot exist two edges (i.e., a request and a grant edge) between the two nodes (one process and one resource) that could form a cycle. Now consider case (ii) where two processes and one

resource exist. Case (ii) can have a path among them, but they cannot form a cycle because according to the bipartite property of a RAG there cannot exist an edge between the two processes. Similarly, case (iii) where one process and two resources exist cannot form a cycle because there cannot exist an edge between the two resources. Thus, forming a cycle must require at least two distinct processes and two distinct resources. Therefore, forming the smallest cycle requires at least four nodes. Furthermore, forming a cycle with four nodes requires at least four edges since otherwise all four nodes cannot be connect nodes. Therefore, the number of edges involved in the smallest cycle is also four. ■

Corollary 2 *In a system state γ_{ij}, the number of edges in any path using all nodes in the smallest possible cycle is three.*

Proof: According to Corollary 1, the smallest possible cycle has four nodes, i.e., two distinct processes and two distinct resources. Let the two processes be p_1, p_2 and the two resources be q_1, q_2. According to the bipartite property of a RAG, there cannot exist an edge between the two processes or between the two resources. Therefore, one longest path is $\{p_1, q_1, p_2, q_2\}$ since this path uses all nodes in γ_{ij}. This path has three edges. There are three more cases of the longest path, and the number of edges in all three cases is also three. Thus, the number of edges in any longest path in the smallest possible cycle in a system γ_i in state γ_{ij} is three. ■

Theorem 2 *If a system γ_i in state γ_{ij} contains a cycle C, then no nodes in cycle C can be excluded from further consideration through any sequence of terminal reduction steps (Definition 29). As a result, cycle C* ***cannot*** *be removed through any sequence of terminal reduction steps. That is, the system state γ_{ij}* ***cannot*** *be completely reduced (Definition 32).*

Proof: A node can be excluded from further consideration by a terminal reduction step only if after the terminal reduction step, the node does not have any edges. However, every node in cycle C is a connect node and thus must have an incoming edge from another node

in cycle C and an outgoing edge to another node in cycle C. That is, none of the edges in cycle C are terminal edges since they are all exclusively connected to connect nodes. Now, according to Definition 29, a terminal reduction step only removes terminal edges; thus, since none of the edges in cycle C are terminal edges, no edge in cycle C can be removed by the first terminal reduction step. Since no edges in cycle C are removed by the first terminal reduction step, all nodes in cycle C remain connect nodes. Thus, for the second terminal reduction step in any sequence, each edge in cycle C remains connected to connect nodes on both ends of the edge. Continuing in this way, we conclude that no edge in cycle C can be removed by any sequence of terminal reduction steps. Hence, since no edge in cycle C can be removed, no node in cycle C can be excluded from further consideration. Furthermore, since neither nodes nor edges in cycle C can be removed, cycle C itself cannot be removed. Therefore, according to Definition 32, γ_{ij} cannot be completely reduced. ■

Lemma 1 *Given system γ_i in state γ_{ij} with cycle C, removing terminal edges (i.e., edges connected to terminal nodes) will not alter cycle C.*

Proof: Every node in cycle C is a connect node. Furthermore, every node in cycle C must have an edge **to** another node in cycle C and **from** another node in cycle C. Therefore, if a node in cycle C has an edge to or from a terminal node, the terminal node cannot be in cycle C. Thus, the removal of an edge to or from a terminal node leaves cycle C intact since none of the edges from a node in cycle C to other nodes in cycle C are edges to or from terminal nodes. ■

Consider a sequence of terminal reductions steps (Definition 29) applied to a given γ_{ij}, resulting in an irreducible system state $\gamma_{i,j+k}$. According to the definition of **irreducible** (Definition 31), $\gamma_{i,j+k}$ has no **terminal** edges, resulting in two cases: (i) $\gamma_{i,j+k}$ is completely reduced or (ii) $\gamma_{i,j+k}$ is incompletely reduced (Definition 32). We will next prove that in case (i) γ_{ij} does not have a deadlock while in case (ii) γ_{ij} has a deadlock.

Lemma 2 *If a system state γ_{ij} can be completely reduced, then it does not have a deadlock.*

Proof: A complete reduction deletes all edges including all request edges. Since a request edge can be deleted only if the request could be fulfilled within a finite time (note that we assume no processes whatsoever take infinite time, i.e., all processes terminate within a finite time), deleting all the request edges implies that all processes can eventually obtain the resources that they have requested. As a result, all the processes can make progress (Definition 33). This fact violates the deadlock definition (Definition 1); hence, γ_{ij} does not have a deadlock.[1] ■

Theorem 3 *If a system state γ_{ij}* ***cannot*** *be completely reduced, then the system contains at least one cycle.*

Proof: If γ_{ij} cannot be completely reduced, then a sequence of reduction steps applied to γ_{ij} results in irreducible state $\gamma_{i,j+k}$ with the property $E(\gamma_{i,j+k}) \neq \emptyset$. In other words, $\gamma_{i,j+k}$ is irreducible and has some edges.

We next note that all the nodes connected to edges in $\gamma_{i,j+k}$ must be connect nodes (since $\gamma_{i,j+k}$ does not contain any terminal nodes). Consider an arbitrary connect node v_ϕ in $\gamma_{i,j+k}$. Taking v_ϕ's outgoing edge (there must be at least one outgoing edge from v_ϕ since v_ϕ is a connect node) we arrive at node $v_{\phi+1}$. Please note that $v_{\phi+1} \neq v_\phi$ since our system does not have any edges from a node back to the same node. Please note also that the edge we took to arrive at $v_{\phi+1}$ is an edge incoming to $v_{\phi+1}$. However, since $v_{\phi+1}$ must also be a connect node, an edge must be outgoing from $v_{\phi+1}$. Taking this outgoing edge, we arrive at another node. Continuing in this way, every node must be connected to another node distinct from itself. Eventually, we arrive either (i) at a node $v_{\phi+m}$ that was previously visited already, or else (ii) at the last node in the graph. In case (i), we have a cycle. In case (ii), this last node in the graph must be a connect node. Since all the nodes have already been visited, the outgoing edge of this last node must lead to a node already

[1] Lemma 2 is equivalent to Corollary 1 on page 189 of [20].

previously visited. Thus, in case (ii) we have a cycle as well. Therefore, if a system state γ_{ij} is not completely reducible, then the system contains at least one cycle. ■

Lemma 3 *If no cycle exists in a system state γ_{ij}, then γ_{ij} can be completely reduced.*

Proof: This lemma is the contraposition of Theorem 3, and it is well-known that the contraposition of a proposition is always true provided that the given proposition is true. That is, if a system state γ_{ij} cannot be completely reduced, then γ_{ij} contains at least one cycle, which implies that if no cycle exists in γ_{ij}, then the system state γ_{ij} is completely reducible.

■

Lemma 4 *In a system γ_i in state γ_{ij}, a process p_k that is making progress cannot be involved in deadlock.*

Proof: Given a system γ_i in state γ_{ij}, if a process p_k is making progress, then according to Definition 33, one of two cases may result: (i) p_k does not need any resources, or (ii) p_k has some resources. In case (i), if p_k is making progress, and it does not need any resources, then according to the definition of deadlock (Definition 1), p_k has nothing to do with deadlock. In case (ii), if p_k holding some resources is making progress, then, according to Assumption 10, p_k has all required resources, will finish using the resources, and will then release the resources; thus, p_k has, at this instant, no unfulfilled resource requests preventing its progress (and eventual release of the resources it does hold). Therefore, p_k does not fulfill Condition 4 (see Section 1.3); thus, p_k cannot be involved in deadlock. As a result, in both cases (i) and (ii), p_k, which is making progress, is not involved in deadlock. ■

Lemma 5 *If system state γ_{ij} in system γ_i does **not** have a deadlock, then all processes in the system state can **make progress** either now or at some time a finite distance in the future.*

Proof: According to the definition of deadlock (Definition 1), even if one process is blocked while waiting for requirements that can never be satisfied, the system state has

a deadlock. Therefore, unless γ_{ij} has a deadlock, no process exists that is unable to make progress within a finite time. In other words, if γ_{ij} does not have a deadlock, all processes in γ_{ij} must be able to make progress within a finite time. ■

Please note that when a process in a system γ_i in state γ_{ij} acquires a resource for which the process has waited, the corresponding request edge is removed and changed to a grant edge. Accordingly, as requests are fulfilled, request edges are replaced with grant edges. After using the granted resources, a process will eventually release the resources that it has used. Since releasing a resource is expressed as removing a grant edge, a process that releases all of its resources will lose all of its grant edges.

Theorem 4 *Given system γ_i in state γ_{ij} and under Assumptions 1-10, a **cycle** is a necessary and sufficient condition for deadlock.*

Proof: We prove this theorem by contradiction. Suppose that γ_{ij} has a cycle but does not have a deadlock. If γ_{ij} does not have a deadlock, then from Lemma 5 all processes in γ_{ij} can make progress either now or at some time a finite distance in the future. According to Definition 33 and Assumption 10, a process that requires some resources can only make progress when the process obtains all the resources for which the process is waiting. Thus, all the processes being able to make progress within a finite time implies that all processes will (eventually) obtain all needed resources, finish using them, and release them within a finite time. In other words, considering the RAG representation, as processes receive resources, request edges will be changed into corresponding grant edges, and then all the grant edges will eventually be removed as processes release resources. Thus, after all computations are finished, there will exist no edges in the final state $\gamma_{i,j+k}$. Please note that, as stated in Assumption 9, we are considering the case in which no new requests come in during the completion of all computations implied by state γ_{ij}. Since there are no edges after all computations are finished, and since by Theorem 2 a cycle that once exists cannot disappear through terminal reduction steps, there cannot exist a cycle in γ_{ij} at all. This

contradicts our supposition that γ_{ij} has a cycle but does not have a deadlock. Therefore, if γ_{ij} has a cycle, then γ_{ij} has a deadlock.

Now assume that γ_{ij} has no cycle but has a deadlock. If γ_{ij} has no cycle, then according to Lemma 3, γ_{ij} is completely reducible. This complete reduction indicates no deadlock according to Lemma 2. Therefore, according to Lemmas 2 and 3, γ_{ij}, which has no cycle, cannot have any deadlock, which, however, contradicts our assumption that γ_{ij} has a deadlock. Thus, if γ_{ij} has a deadlock, then at least one cycle must exist in γ_{ij}. As a result, a **cycle** is a necessary and sufficient condition for deadlock. ■

Although it has already been proven in [8] and [20] that a cycle is a necessary and sufficient condition for deadlock, none of the proofs were exactly applicable to our system model with a DDU. Therefore, rather than adapting our system model and notation to prior proofs, we decided to prove Theorem 4 in the exact context that we wanted for our claims of correctness for the proposed PDDA and its hardware implementation in the DDU.

3.2.2 Matrix Representation of a RAG

So far, we have covered terms and properties that are applied to PDDA. As PDDA dramatically reduces deadlock detection time when implemented in hardware, to easily accommodate such hardware, the RAG corresponding to state γ_{ij} is mapped into a matrix M_{ij} that will have exactly the same request and grant edges as the RAG corresponding to γ_{ij}. However, M_{ij} will utilize a slightly different notation for each edge. We now define a RAG matrix M and a terminal reduction step applied to a matrix M before introducing an algorithm that exploits the matrix representation.

Definition 34 *The purpose of this definition is to define matrices that correspond to graph γ, system γ_i and state γ_{ij} from Definitions 15 and 17. A **RAG matrix M** is a matrix mapped from a RAG γ and represents an arbitrary system with processes and resources. A **system matrix M_i** is defined as a matrix representation of a particular system γ_i, where the rows (fixed in size) of matrix M_i represent the fixed set Q of resource nodes of γ_i, and the columns*

(fixed in size) of matrix M_i represent the fixed set P of process nodes of γ_i. We denote another notation of this relationship as $M_i \equiv \gamma_i$ for the sake of simplicity. A ***state matrix*** $\boldsymbol{M_{ij}}$ *is a matrix that represents a particular system state γ_{ij}, i.e., $M_{ij} \equiv \gamma_{ij}$. Edges in system state γ_{ij} are mapped into the corresponding array elements using the following rule:*

Given $E = \{R \cup G\}$ from γ_{ij},

$$\boldsymbol{M_{ij}} = \begin{bmatrix} \alpha_{11} & \alpha_{12} & \dots & \alpha_{1n} \\ \alpha_{21} & \alpha_{22} & \dots & \alpha_{2n} \\ : & : & \alpha_{st} & : \\ \alpha_{m1} & \alpha_{m2} & \dots & \alpha_{mn} \end{bmatrix},$$

for all rows $1 \leq s \leq m$ and for all columns $1 \leq t \leq n$:

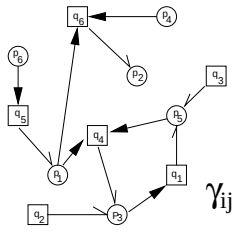
$\alpha_{st} = g_{s \rightharpoonup t}$ *(or simply 'g'), if there exists a grant edge $(q_s, p_t) \in G$*

$\alpha_{st} = r_{t \rightarrow s}$ *(or simply 'r'), if there exists a request edge $(p_t, q_s) \in R$*

$\alpha_{st} = 0_{st}$ *('0' or a blank space), otherwise.*

Example 10 State Matrix Representation

The system in state γ_{ij} shown in Figure 10(a) can be represented in the matrix form shown in (b). For the sake of better understanding, we will use the matrix representation shown in Figure 10 (c) from now on. ■



γ_{ij}

$$\boldsymbol{M_{ij}} = \begin{bmatrix} 0 & 0 & r & 0 & g & 0 \\ 0 & 0 & g & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & g & 0 \\ r & 0 & g & 0 & r & 0 \\ g & 0 & 0 & 0 & 0 & r \\ r & g & 0 & r & 0 & 0 \end{bmatrix} \Longrightarrow$$

M_{ij}	p_1	p_2	p_3	p_4	p_5	p_6
q_1			r		g	
q_2			g			
q_3					g	
q_4	r		g		r	
q_5	g					r
q_6	r	g		r		

(a) (b) (c)

Figure 10: Matrix representation example.

Definition 35 *A state matrix M_{ij} is said to be* **reducible** *if its corresponding system state γ_{ij} is reducible (Definition 31). Similarly, M_{ij} is said to be* **irreducible** *if its corresponding γ_{ij} is irreducible.*

Definition 36 *A system matrix $M_{i,j+k}$ is said to be* ***completely reduced*** *if its corresponding system state $\gamma_{i,j+k}$ is completely reduced (Definition 32). Likewise, $M_{i,j+k}$ is said to be* **incompletely reduced** *if its corresponding $\gamma_{i,j+k}$ is incompletely reduced.*

Definition 37 *A* ***terminal row*** *is a row r_s (recall that row r_s corresponds to resource q_s) of matrix M_{ij} such that either (i) all non-zero entries $\{\alpha_{st_r} \neq 0, 1 \leq t_r \leq n\}$ are request entries $r_{t_r \rightarrow s}$ with at least one request entry (i.e., one or more request entries and no grant entry in the row) or (ii) one entry α_{st_g}, $1 \leq t_g \leq n$, is a grant $g_{s \rightarrow t_g}$ with the rest of the entries $\{\alpha_{st}, 1 \leq t \leq n, t \neq t_g\}$ equal to zero.*

Definition 38 τ_{r_s} *represents an evaluation of whether or not a row is a terminal row. That is, if τ_{r_s} is true (i.e., '1'), the corresponding row r_s is a terminal row; otherwise, if τ_{r_s} is false (i.e., '0'), the corresponding row r_s is not a terminal row.*

Example 11 Terminal Row

In Figure 11, row q_2 is a terminal row according to case (i) of Definition 37. Also, row q_5 is another terminal row, this time according to case (ii) of Definition 37. ■

M_{ij}	p_1	p_2	p_3	p_4	p_5	p_6
q_1						
q_2			r		r	
q_3						
q_4	r		g			
q_5	g					
q_6						

Figure 11: Terminal row example.

Please note that for a terminal row r_s, its corresponding resource node q_s is a terminal node; thus, all non-zero entries in row r_s are terminal edges. Note also the effect of

Assumption 3 stated in Section 1.5 that a resource can only be granted to one process. Specifically, the effect is that in case (ii) in Definition 37, a row (corresponding to a resource) may have at most one grant entry in the row.

Definition 39 *A **terminal column** is a column c_t (recall that column c_t corresponds to process p_t) of matrix M_{ij} such that either (i) all non-zero entries $\{\alpha_{st} \neq 0, 1 \leq s \leq m\}$ are request entries with at least one request entry (i.e., one or more request entries and no grant entry in the column) or (ii) all non-zero entries $\{\alpha_{st} \neq 0, 1 \leq s \leq m\}$ are grant entries with at least one grant entry (i.e., one or more grant entries and no request entry in the column).*

Definition 40 τ_{c_t} *represents an evaluation of whether or not a column is a terminal column. That is, if τ_{c_t} is true (i.e., '1'), the corresponding column c_t is a terminal column; otherwise, if τ_{c_t} is false (i.e., '0'), the corresponding column c_t is not a terminal column.*

Example 12 Terminal Column

In Figure 12, column p_5 is a terminal column according to case (i) of Definition 39. Also, column p_3 is another terminal column, this time according to case (ii) of Definition 39. ■

M_{ij}	p_1	p_2	p_3	p_4	p_5	p_6
q_1					r	
q_2			g			
q_3						
q_4	r		g		r	
q_5	g					
q_6						

Figure 12: Terminal column example.

Please note that in a terminal column c_t, its corresponding process node p_t is a terminal node, and all non-zero entries in the terminal column c_t are terminal edges.

Definition 41 *Given state matrix M_{ij}, function $T_s(M_{ij})$ produces the on-set (i.e., true set) of all terminal rows.*

Definition 42 *Given state matrix M_{ij}, function $T_t(M_{ij})$ produces the on-set of all terminal columns.*

Example 13 Production of terminal rows and columns

In Figure 11, $T_s(M_{ij}) = \{\tau_{r_2}, \tau_{r_5}\}$, and in Figure 12, $T_t(M_{ij}) = \{\tau_{c_3}, \tau_{c_5}\}$. ■

Definition 43 ***A terminal reduction step ϵ*** *is a unary operator $\epsilon : M_{ij} \mapsto M_{i,j+1}$, where ϵ calculates the terminal edge set $\tau(M_{ij}) \equiv \tau(\gamma_{ij})$ defined in Definition 20 and returns $M_{i,j+1}$ such that all terminal edges $\tau(M_{ij})$ found are removed by setting the terminal entries found to zero; thus, the next iteration $M_{i,j+1}$ will start with equal or fewer total edges as compared to M_{ij}. This terminal reduction step is denoted as $\epsilon(M_{ij})$. The formula for $M_{i,j+1} = \epsilon(M_{ij})$ is shown in Equation 1 (see Definition 34 for the meaning of $M_{i,j+1} \equiv \gamma_{i,j+1}$):*

$$\begin{array}{rclcl} M_{i,j+1} & = & \epsilon(M_{ij}) & \equiv & \epsilon(\gamma_{ij}) \\ & = & \{V, E(M_{ij}) - \tau(M_{ij})\} & \equiv & \{V, E(\gamma_{ij}) - \tau(\gamma_{ij})\} \quad = \quad \gamma_{i,j+1} \end{array} \tag{1}$$

Please note that the removals of terminal edges in M_{ij} enable the discovery of new terminal nodes in $M_{i,j+1}$. Any new terminal nodes that appear in $M_{i,j+1}$ were connect nodes in M_{ij} that were connected to terminal nodes in M_{ij}.

Example 14 One Step of Terminal Reduction (ϵ)

Figure 13(b) shows a new matrix $M_{i,j+1}$ after the matrix reduction step ϵ defined in Definition 43 is applied to M_{ij} shown in Figure 13 (a). In matrix M_{ij}, q_2 and q_3 are terminal nodes by Definition 18 (also terminal rows by Definition 37); thus, all the edges in these rows are terminal edges by Definition 19. Therefore, all the edges in rows q_2 and q_3 can be removed. Likewise, p_2, p_4 and p_6 are terminal nodes (also terminal columns by Definition 39); hence, all edges in these columns can be removed, resulting in matrix $M_{i,j+1}$ shown in Figure 13 (b). ■

Definition 44 ***A terminal reduction sequence ξ****, applicable to a matrix M_{ij}, is a sequence of k terminal reduction steps ϵ (recall that ϵ is a terminal reduction step) such that (i) $M_{ij} \mapsto M_{i,j+1} \mapsto \cdots \mapsto M_{i,j+k}$; (ii) $M_{i,j+k}$ is irreducible; and (iii) $\{M_{i,j+h}, 0 \leq h <$*

M_{ij}	p_1	p_2	p_3	p_4	p_5	p_6
q_1			r		g	
q_2			g			
q_3					g	
q_4	r		g		r	
q_5	g					r
q_6	r	g		r		

(a)

$\epsilon \Longrightarrow$

$M_{i,j+1}$	p_1	p_2	p_3	p_4	p_5	p_6
q_1			r		g	
q_2						
q_3						
q_4	r		g		r	
q_5	g					
q_6	r					

(b)

Figure 13: Terminal reduction step (ϵ) example.

$k\}$ *are all reducible. A terminal reduction sequence is called a* ***complete reduction*** *when the sequence of terminal reduction steps corresponding to* ξ *results in* $M_{i,j+k}$ *such that the irreducible state matrix* $M_{i,j+k}$ *contains all zero entries (note that this means that* $\gamma_{i,j+k}$ *corresponding to* $M_{i,j+k}$ *has no edges:* $E(\gamma_{i,j+k}) = \emptyset$*). A terminal reduction sequence is called an* ***incomplete reduction*** *when* ξ *returns* $M_{i,j+k}$ *with at least one non-zero entry (note that this means that* $\gamma_{i,j+k}$ *corresponding to* $M_{i,j+k}$ *has at least one edge:* $E(\gamma_{i,j+k}) \neq \emptyset$*). Another representation of a terminal reduction sequence is shown in Equation 2.*

$$\begin{aligned} M_{i,j+k} &= \xi(M_{ij}) \\ &= \epsilon^{(k)}(\ldots\epsilon^{(2)}(\epsilon^{(1)}(M_{ij}))\ldots) \\ &= \epsilon(\ldots\epsilon(\epsilon(M_{ij}))\ldots) \end{aligned} \tag{2}$$

3.2.3 Parallel Deadlock Detection Algorithm (PDDA)

In this section, we first introduce a terminal reduction algorithm which implements $\boldsymbol{\xi}$ (Definition 44). We next show PDDA which uses the terminal reduction algorithm. Finally, we prove the correctness of PDDA and PDDA's run-time complexity when implemented in parallel hardware (i.e., the DDU).

Algorithm 1 is an implementation of the terminal reduction sequence $\boldsymbol{\xi}$ shown in Definition 44. We summarize the operation of Algorithm 1. Lines 2 and 3 of Algorithm 1 initialize two variables: iterator k and matrix M_{iter} that is initially a copy of input matrix M_{ij}. Line 5 finds all terminal rows (Definition 37), and Line 6 finds all terminal columns

(Definition 39). Line 7 checks whether or not M_{iter} is reducible further (Definition 35). Lines 8 and 9 remove all terminal edges found at the current iteration. On the whole, the terminal reduction step $\epsilon(M_{ij})$ of Definition 43 corresponds to Lines 5-9 of Algorithm 1, which iterates until the matrix M_{iter} becomes irreducible; this iteration process implements the terminal reduction sequence $\boldsymbol{\xi}$. Please note that, in hardware implementation, Lines 5 and 6 of Algorithm 1 are executed at the same time in parallel, as are Lines 8 and 9.

Algorithm 1 Terminal Reduction Algorithm

```
1   ξ(M_ij) {
2       k = 0;
3       M_iter = M_ij;
4       while (1) {
            /* parallel on */
5           calculate T_s(M_iter); /* determine all terminal rows */
6           calculate T_t(M_iter); /* determine all terminal columns */
            /* parallel off */
7           if ((T_s(M_iter) = ∅) and (T_t(M_iter) = ∅)) break; /* if no more terminals */
            /* parallel on */
8           for each terminal row ∈ T_s(M_iter), set all entries in τ_{r_s} to zero;
9           for each terminal column ∈ T_t(M_iter), set all entries in τ_{c_t} to zero;
            /* parallel off */
10          k = k + 1;
11      }
12      M_{i,j+k} = M_iter;
13      return M_{i,j+k};
14  }
```

Algorithm 2 Parallel Deadlock Detection Algorithm (PDDA)

```
1   Deadlock_Detect_Matrix (γ_ij) {
2       M[s, t] = [α_st], where
3           s = 1, ... , m and t = 1, ... , n
4           α_st = r, if ∃(p_t, q_s) ∈ E(γ_ij)
5           α_st = g, if ∃(q_s, p_t) ∈ E(γ_ij)
6           α_st = 0, otherwise.
7       M_{i,j+k} = ξ(M_ij); /* call Algorithm 1 */
8       if (M_{i,j+k} == [0]) { /* matrix of all zeros */
9           return 0; /* no deadlock */
10      } else {
11          return 1; /* deadlock detected */
12      }
13  }
```

We now summarize the operation of Algorithm 2 (i.e., PDDA). Lines 2-6, given γ_{ij}, construct the corresponding matrix M_{ij} according to Definition 34. Next, Line 7 calls Algorithm 1 with argument M_{ij}. When Algorithm 1 is completed, Lines 8-12 of Algorithm 2 determine whether γ_{ij} has a deadlock or not by considering returned matrix $M_{i,j+k}$: if $M_{i,j+k}$ is empty, the corresponding γ_{ij} has no deadlock; otherwise, deadlock(s) exist. Finally, Algorithm 2 returns '1' if the system state under consideration has deadlock(s), or '0' if no deadlock. Please note that Algorithm 2, which includes Algorithm 1, is referred to as PDDA. Next, we present a simple example that shows results at each iteration of PDDA.

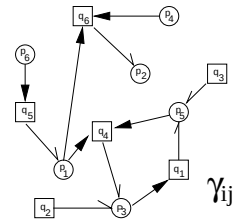
Example 15 Matrix Reduction Sequence and Deadlock Detection

Consider the system state γ_{ij} shown in Figure 10 again. Figure 14 illustrates how each step of Algorithm 1 is applied to matrix M_{ij} and its RAG representation γ_{ij}. The original γ_{ij} and M_{ij} are shown at step 0 in Figure 14(a). After one iteration of lines 5-10 in Algorithm 1, the state becomes Step 1 ($M_{i,j+1}$) shown in (b). One more iteration produces step 2 ($M_{i,j+2}$) shown in (c). After the third iteration, Algorithm 1 returns an irreducible matrix $M_{i,j+3}$ shown in (d); thus, Algorithm 2 detects deadlock by evaluating the returned matrix $M_{i,j+3}$. ■

3.2.4 Proof of the Correctness of PDDA

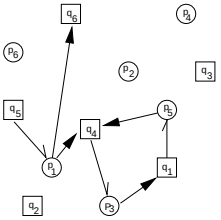
Theorem 5 *PDDA detects deadlock if and only if there exists a cycle in state* γ_{ij}.

Proof: Consider M_{ij} corresponding to γ_{ij} (Definition 34). (a) Algorithm 1 returns, by construction, an irreducible matrix $M_{i,j+k}$. (b) By the definition of **irreducible** (Definition 35), $M_{i,j+k}$ has no terminal edges, yielding two cases: (i) $M_{i,j+k}$ is completely reduced, or (ii) $M_{i,j+k}$ is incompletely reduced (Definition 36). In case (i), by Lemma 2, M_{ij} (i.e., γ_{ij}) has no deadlock. In case (ii), by Theorem 3, M_{ij} (i.e., γ_{ij}) has at least one cycle; thus, by Theorem 4, γ_{ij} has a deadlock. ■



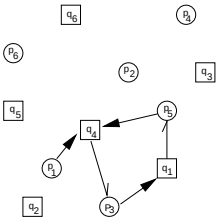
M_{ij}	p_1	p_2	p_3	p_4	p_5	p_6
q_1			r		g	
q_2			g			
q_3					g	
q_4	r		g		r	
q_5	g					r
q_6	r	g		r		

(a) step 0



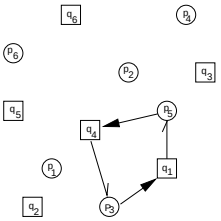
$M_{i,j+1}$	p_1	p_2	p_3	p_4	p_5	p_6
q_1			r		g	
q_2						
q_3						
q_4	r		g		r	
q_5	g					
q_6	r					

(b) step 1



$M_{i,j+2}$	p_1	p_2	p_3	p_4	p_5	p_6
q_1			r		g	
q_2						
q_3						
q_4	r		g		r	
q_5						
q_6						

(c) step 2



$M_{i,j+3}$	p_1	p_2	p_3	p_4	p_5	p_6
q_1			r		g	
q_2						
q_3						
q_4			g		r	
q_5						
q_6						

(d) step 3

Figure 14: A sample sequence of reduction steps.

3.2.5 Proof of the Run-time Complexity of the DDU

Lemma 6 *In a RAG γ_{ij}, an upper bound on the number of edges in a path is $2 \times min(m, n)$, where m is the number of resources and n is the number of processes.*

Proof: By Definitions 11 and 15, RAG γ_{ij} is a bipartite graph in which any request edge spans from a process node to a resource node and any grant edge spans from a resource node to a process node. That is, γ_{ij} cannot have any edge from process p_g to process p_h for any two processes p_g, $p_h \in V(\gamma_{ij})$. Similarly, γ_{ij} cannot have any edge from resource q_k to resource q_l for any two resources q_k, $q_l \in V(\gamma_{ij})$. Also recall that every node in each path of γ_{ij} is distinct by Definition 24.

Let us consider the following three possibilities: (i) $m = n$, (ii) $m > n$, or (iii) $m < n$. For case (i), where m equals n, one longest path is $\{p_1, q_1, p_2, q_2, \ldots, p_m, q_m\}$ since this path uses all the nodes in γ_{ij}, and since every node in a path must be distinct (i.e., every node can only be listed once). In this case, the number of edges involved in the path is $2 \times m - 1$ (or $2 \times n - 1$). For case (ii), where m is greater than n (i.e., $m - n > 0$), one longest path is $\{q_1, p_1, q_2, p_2, \ldots, q_n, p_n, q_{n+1}\}$; this path cannot be lengthened since every node in a path must be distinct, and since all n process-nodes are already used in the path. Therefore, the number of edges in this path is $2 \times n$. Likewise, for case (iii), where n is greater than m (i.e., $n - m > 0$), the number of edges involved in any longest path is $2 \times m$.

As a result, cases (i), (ii) and (iii) show that the number of edges of the maximum possible longest path in a RAG γ_{ij} is $2 \times min(m, n)$. ■

Theorem 6 *Algorithm 1, when implemented in parallel hardware, completes its computation in at most* $2 \times min(m, n) - 3 = O(min(m, n))$ *steps, where* m *is the number of resources and* n *is the number of processes.*

Proof: Given M_{ij}, we consider the corresponding γ_{ij}, which has a one-to-one correspondence with M_{ij} according to Definition 34. At each iteration in Line 7 of Algorithm 1, if Line 7 evaluates to false, then there exists at least one terminal row or column; thus, another iteration needs to continue. Please note that the worst case number of iterations will occur when γ_{ij} has the longest reducible path. However, the simple longest reducible path cannot give the worst case because such a simple path (see Definition 25) has a terminal node at

each end of the simple path; thus, each ϵ reduction will remove two edges at a time! Thus, maximizing the number of iterations requires that one end of the longest path be a terminal node while the other end of the longest path connects to a cycle, preventing this end from being reduced.

Now consider the case where γ_{ij} has the longest possible path that is connected to a cycle at one end of the path. According to Lemma 6, since the number of edges of the maximum possible longest path in a bipartite RAG has an upper bound of $2 \times min(m, n)$, the number of iterations in the worst case is bound by $2 \times min(m, n)$. Furthermore, since one end of the path is connected to a cycle, the number of iterations will further be reduced by the path edges contained in the cycle (Definition 27) since edges in a cycle can never be terminal edges. Accordingly, the worst case (i.e., the maximum number of iterations) occurs when the size of the cycle is at its minimum. According to Corollary 2, when all the nodes in the smallest possible cycle are used, the longest path has three edges in this smallest possible cycle. Therefore, in the worst case, $2 \times min(m, n) - 3$ is an upper bound on the number of edges in the longest possible path that are not also part of a cycle.

Hence, the number of iterations required to reach an irreducible state becomes at most $2 \times min(m, n) - 3 = O(min(m, n))$ in the worst case. ■

The next example shows terminal reduction steps (Algorithm 1) and a run-time complexity calculation for a simple path case.

Example 16 Run-time Complexity Calculation in a Simple Path Case

M_{ij}	p_1	p_2	p_3	p_4	p_5	p_6
q_1	g	r				
q_2		g	r			
q_3			g	r		
q_4				g	r	
q_5					g	r

Figure 15: Simple path example.

A RAG for the matrix shown in Figure 15 is

$$p_1 \leftarrow q_1 \leftarrow p_2 \leftarrow q_2 \leftarrow p_3 \leftarrow q_3 \leftarrow p_4 \leftarrow q_4 \leftarrow p_5 \leftarrow q_5 \leftarrow p_6$$

Due to the two terminal nodes at both ends, after the first iteration of Algorithm 1, the RAG becomes

$$q_1 \leftarrow p_2 \leftarrow q_2 \leftarrow p_3 \leftarrow q_3 \leftarrow p_4 \leftarrow q_4 \leftarrow p_5 \leftarrow q_5$$

After the second iteration of Algorithm 1, the RAG becomes

$$p_2 \leftarrow q_2 \leftarrow p_3 \leftarrow q_3 \leftarrow p_4 \leftarrow q_4 \leftarrow p_5$$

After the fourth iteration of Algorithm 1, the RAG becomes

$$p_3 \leftarrow q_3 \leftarrow p_4$$

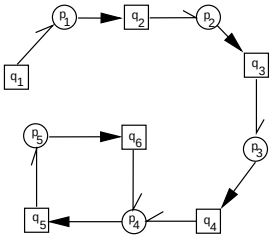
which, after the fifth iteration, leaves no edges.

As shown in this example, in general, the number of iterations required to reach an irreducible state in a simple path case is $min(m, n)$, which, in this case, is 5. ■

The next example illustrates one of the worst cases of terminal reduction.

Example 17 Run-time Complexity Calculation in the Worst Case

In the dangling path shown in Figure 16, it takes 7 iterations to remove all reducible terminal edges. The size of the cycle shown in Figure 16 is four (i.e. the minimum possible). Thus, according to Theorem 6, the number of iterations to reach an irreducible state for the RAG shown in Figure 16 is $(2 \times min(m, n) - 3)$; i.e., $(2 \times 5 - 3)$ = 7 iterations. ■



(a) a RAG

M_{ij}	p_1	p_2	p_3	p_4	p_5
q_1	g				
q_2	r	g			
q_3		r	g		
q_4			r	g	
q_5				r	g
q_6				g	r

(b) M_{ij}

Figure 16: Dangling path connected to a cycle when $m \neq n$.

Examples 16 and 17 demonstrate for specific cases that Algorithm 1 has a worst-case run-time complexity of $2 \times min(m,n) - 3 = O(min(m,n))$ when PDDA is implemented in hardware and executed in parallel.

In this section we have proven the correctness and run-time complexity of the DDU. We now aim to describe the operation of the DDU in great detail using matrix representations and Boolean algebra.

3.3 Hardware Implementation of PDDA

In this section, we will explain the operation of PDDA in great detail, considering that PDDA is to be implemented in hardware (the DDU). In Section 3.3.1, we will explain a series of logical operations performed on M_{ij} more generally in terms of matrix theory. Next, in Section 3.3.2, we will provide two detailed examples to demonstrate each logical operation of PDDA using this matrix theory and then describe the architecture of the DDU in detail. To see the examples before the theoretical descriptions, readers may go directly to Section 3.3.2.

3.3.1 Step-by-step Operations of the DDU with Mathematical Representations

As described in Section 3.1.1, when the DDU is informed of or notices an event of either a request or a grant, a series of logic operations occurs: (i) finding terminal nodes, (ii) reducing terminal edges, (iii) iterating (i) and (ii) until no more terminal edges exist, and (iv) checking for deadlock.

A given system state γ_{ij} is equivalently represented by a system state matrix M_{ij} so that, based on M_{ij}, the DDU performs the sequence of operations shown in Algorithms 1 and 2 in Section 3.2.3 and determines whether or not the given state has a deadlock. In the general case, a system state matrix can be explicitly represented as shown in Equation 3.

$$\boldsymbol{M_{ij}} = \begin{bmatrix} \alpha_{11} & \alpha_{12} & \dots & \alpha_{1t} & \dots & \alpha_{1n} \\ \alpha_{21} & \alpha_{22} & \dots & \alpha_{2t} & \dots & \alpha_{2n} \\ : & : & : & : & : & : \\ \alpha_{s1} & \alpha_{s2} & \dots & \alpha_{st} & \dots & \alpha_{sn} \\ : & : & : & : & : & : \\ \alpha_{m1} & \alpha_{m2} & \dots & \alpha_{mt} & \dots & \alpha_{mn} \end{bmatrix} = M_{iter} \text{ (at the first iteration)} \quad (3)$$

where m is the number of resources and n is the number of processes.

We now explain the logical operation of the DDU circuit in detail using Boolean algebra and matrix computation. Lines 2-6 of Algorithm 2 illustrate the translation from a given system state γ_{ij} into its matrix form M_{ij}. Each matrix element α_{st} represents one of the following: $g_{s\rightarrow t}$ (a grant edge), $r_{t\rightarrow s}$ (a request edge) or 0_{st} (no edge). That is, α_{st} is ternary-valued. The hardware implementation of PDDA in digital logic requires that the values of each element be represented in a binary format with a minimum number of encoding bits, preferably. Since α_{st} is ternary-valued, α_{st} can be minimally defined as a pair of two bits $\alpha_{st} = (\alpha_{st}^r, \alpha_{st}^g)$. If an entry α_{st} is a grant edge g, bit α_{st}^r is set to 0, and α_{st}^g is set to 1; if an entry α_{st} is a request edge r, bit α_{st}^r is set to 1, and α_{st}^g is set to 0; otherwise, both bits α_{st}^r and α_{st}^g are set to 0. Hence, an entry α_{st} can be only one of the following binary encodings: 01 (a grant edge), 10 (a request edge) or 00 (no activity). This way of element representation is a variant of the positional cube notation developed for logic minimization of digital circuits.[2] Please note that $\alpha_{st} = (\alpha_{st}^r, \alpha_{st}^g) = 11$ never appears (i.e., is illegal).

Next, Line 7 of Algorithm 2 calls Algorithm 1. Please note that since this process occurs in hardware, an actual call will not occur. Instead of the call in Algorithm 1, the DDU just starts operation with M_{ij}; that is, M_{ij} just becomes M_{iter} during iterations, as shown in Line 3 of Algorithm 1. However, the original M_{ij} is not altered; instead each output of element α_{st} may temporarily be suppressed as an emulation of edge reduction (the exact

[2] [9] and Chapter 2 of [10]

hardware technique used to achieve this "suppression" will be explained in Section 3.3.4). Then, M_{iter} is processed in row-wise and column-wise directions simultaneously, i.e., two-dimensional operation. The row-wise direction operation corresponds to Lines 5 and 8 of Algorithm 1, which finds all terminal rows in M_{iter} and sets to zero all entries in each terminal row. The column-wise direction operation corresponds to Lines 6 and 9 of Algorithm 1, which finds all terminal columns and sets to zero all entries in each terminal column.

In order to facilitate the two-dimensional operations, we introduce two more matrices, M^c_{iter} and M^r_{iter}. Even though these two matrices are not actually implemented as separate memory locations in hardware (for hardware implementation details, please see Section 3.3.4), we nonetheless represent the two-dimensional operation with two-bit binary encoding column and row representations, M^c_{iter} and M^r_{iter} (which are shown in Equation 4 and Equation 5, respectively), so that understanding parallel two-dimensional operations inside the DDU can become easier.

$$\boldsymbol{M^c_{iter}} = \begin{bmatrix} (\alpha^r_{11}, \alpha^g_{11}) & \dots & (\alpha^r_{1t}, \alpha^g_{1t}) & \dots & (\alpha^r_{1n}, \alpha^g_{1n}) \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ (\alpha^r_{s1}, \alpha^g_{s1}) & \dots & (\alpha^r_{st}, \alpha^g_{st}) & \dots & (\alpha^r_{sn}, \alpha^g_{sn}) \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ (\alpha^r_{m1}, \alpha^g_{m1}) & \dots & (\alpha^r_{mt}, \alpha^g_{mt}) & \dots & (\alpha^r_{mn}, \alpha^g_{mn}) \end{bmatrix} \quad (4)$$

$$\boldsymbol{M^r_{iter}} = \left[\begin{array}{ccccc} \alpha^r_{11} & \ldots & \alpha^r_{1t}, & \ldots & \alpha^r_{1n} \\ \alpha^g_{11} & \ldots & \alpha^g_{1t} & \ldots & \alpha^g_{1n} \\ \hline : & : & : & : & : \\ \hline \alpha^r_{s1} & \ldots & \alpha^r_{st} & \ldots & \alpha^r_{sn} \\ \alpha^g_{s1} & \ldots & \alpha^g_{st} & \ldots & \alpha^g_{sn} \\ \hline : & : & : & : & : \\ \hline \alpha^r_{m1} & \ldots & \alpha^r_{mt} & \ldots & \alpha^r_{mn} \\ \alpha^g_{m1} & \ldots & \alpha^g_{mt} & \ldots & \alpha^g_{mn} \end{array}\right] \tag{5}$$

From matrices M^c_{iter} and M^r_{iter}, finding any terminal edges that exist in the current iteration (which corresponds to Lines 5 and 6 in Algorithm 1) will entail three logical operations performed in sequence: (i) Bit-Wise-Or (BWO) in Equations 6 and 7, (ii) eXclusive-OR (XOR) in Equations 8 and 9, and (iii) OR as shown in Equation 10. All these BWO, XOR and OR logical operations are performed in parallel throughout the matrix. Two parallel BWO operations are derived as shown in Equations 6 and 7.

$$BWO^c_{iter} = \left[\begin{array}{ccccccc} (\alpha^r_{c_1}, \alpha^g_{c_1}) & (\alpha^r_{c_2}, \alpha^g_{c_2}) & \cdots & (\alpha^r_{c_t}, \alpha^g_{c_t}) & \cdots & (\alpha^r_{c_n}, \alpha^g_{c_n}) \end{array}\right] \tag{6}$$

where $\forall c_t, 1 \leq t \leq n$ (i.e., for all columns), $\alpha^r_{c_t} = \bigvee_{s=1}^{m} \alpha^r_{st}$ and $\alpha^g_{c_t} = \bigvee_{s=1}^{m} \alpha^g_{st}$ (notation $\bigvee$ means Bit-Wise-Or of elements).

$$BWO^r_{iter} = \left[\begin{array}{ccccccc} (\alpha^r_{r_1}, \alpha^g_{r_1}) & (\alpha^r_{r_2}, \alpha^g_{r_2}) & \cdots & (\alpha^r_{r_s}, \alpha^g_{r_s}) \cdots & (\alpha^r_{r_m}, \alpha^g_{r_m}) \end{array}\right]^T \tag{7}$$

where $\forall r_s, 1 \leq s \leq m$ (i.e, for all rows), $\alpha^r_{r_s} = \bigvee_{t=1}^{n} \alpha^r_{st}$ and $\alpha^g_{r_s} = \bigvee_{t=1}^{n} \alpha^g_{st}$.

The XOR operations are performed as follows.

$$XOR^c_{iter} = \left[\begin{array}{ccccccc} \tau_{c_1} & \tau_{c_2} & \cdots & \tau_{c_t} & \cdots & \tau_{c_n} \end{array}\right] \tag{8}$$

where $\forall c_t, 1 \leq t \leq n$, $\tau_{c_t} = \alpha^r_{c_t} \oplus \alpha^g_{c_t}$ and $\oplus$ denotes eXclusive-OR. As shown in Definition 40, if the value of τ_{c_t} is true (i.e., '1'), column c_t is a terminal column and all non-zero

entries in column c_t are terminal edges.

$$XOR^r_{iter} = \begin{bmatrix} \tau_{r_1} & \tau_{r_2} & \cdots & \tau_{r_s} & \cdots & \tau_{r_m} \end{bmatrix}^T \tag{9}$$

where $\forall r_s, 1 \leq s \leq m$, $\tau_{r_s} = \alpha^r_{r_s} \oplus \alpha^g_{r_s}$. As shown in Definition 38, if the value of τ_{r_s} is true (i.e., '1'), row r_s is a terminal row and all non-zero entries in row r_s are terminal edges. Thus, Equation 8 shows the existence of terminal nodes in any column, and Equation 9, in any row. As a result, if all entries in both XOR^c_{iter} and XOR^r_{iter} are false (i.e., '0'), there exist no terminal nodes. If, however, one or more entries in either XOR^c_{iter} or XOR^r_{iter} are true (i.e., '1'), there exist terminal node(s) and thus terminal edge(s).

Among the sequence of three logical operations, Equation 10 shows the OR operation that produces the termination condition (i.e., the further reducibility of matrix M_{iter}, which corresponds to Line 7 in Algorithm 1) at each iteration. If T_{iter} is one, i.e., logically true, then more terminal edges exist; thus, further iterations must continue. However, if the current matrix is irreducible (i.e., it has no terminal edges), T_{iter_k} will contain a '0'; thus, further iterations would accomplish nothing.

$$T_{iter} = \tau_C \vee \tau_R \tag{10}$$

where $\tau_C = \bigvee_{t=1}^{n} \tau_{c_t}$ and $\tau_R = \bigvee_{s=1}^{m} \tau_{r_s}$. The next iteration $M_{iter_{k+1}}$ is derived from Equations 8 and 9 according to the following criterion (which corresponds to Lines 8 and 9 of Algorithm 1) considering the current iteration to be M_{iter_k}.

$$(\alpha^r_{st}, \alpha^g_{st})_{k+1} = \begin{cases} (\alpha^r_{st}, \alpha^g_{st})_k, & \text{if } \tau_{c_t} = 0 \text{ and } \tau_{r_s} = 0 \\ (0,0), & \text{if } \tau_{c_t} = 1 \text{ or } \tau_{r_s} = 1 \end{cases} \tag{11}$$

where k refers to k^{th} iteration, and $k+1$ refers to $(k+1)^{\text{th}}$ iteration. That is, the next iteration (if it occurs) will begin with a new matrix $M_{iter_{k+1}}$ calculated by Equation 11 from M_{iter_k}.

Before finishing PDDA, however, one more important process remains: deadlock detection, which requires two more parallel logic operations shown in Equations 12 and 13.

These two parallel logic operations are important when carried out after the last iteration of Equation 10 (i.e., after Equation 10 yields a result of $T_{iter} = 0$). Equation 12 represents the existence of connect nodes involved in a cycle in any column, and Equation 13, in any row.

$$ANDc_{iter} = \left[\begin{array}{ccccccc} \phi_{c_1} & \phi_{c_2} & \cdots & \phi_{c_t} & \cdots & \phi_{c_n} \end{array} \right] \tag{12}$$

where $\forall c_t, 1 \leq t \leq n, \phi_{c_t} = \alpha^r_{c_t} \wedge \alpha^g_{c_t}$ and $\wedge$ denotes bit-wise-and.

$$ANDr_{iter} = \left[\begin{array}{ccccccc} \phi_{r_1} & \phi_{r_2} & \cdots & \phi_{r_s} & \cdots & \phi_{r_m} \end{array} \right]^T \tag{13}$$

where $\forall r_s, 1 \leq s \leq m, \phi_{r_s} = \alpha^r_{r_s} \wedge \alpha^g_{r_s}$ and $\wedge$ denotes bit-wise-and. If the value of ϕ_{c_t} is true (i.e., '1'), column c_t is a connect node, and similarly if the value of ϕ_{r_s} is true, row r_s is a connect node.

Finally, Equation 14 produces the result of deadlock detection, which corresponds to lines 8-12 of Algorithm 2.

$$D_{iter} = \phi_c \vee \phi_r, \qquad \text{when } T_{iter} = 0 \tag{14}$$

where $\phi_c = \bigvee_{t=1}^{n} \phi_{c_t}$ and $\phi_r = \bigvee_{s=1}^{m} \phi_{r_s}$.

We define two more equations that are used to describe the DDU architecture in Section 3.3.4. From Equations 8 and 12, we define a column weight vector as follows:

$$W^c_{iter} = \left[\begin{array}{ccccccc} w_{c_1} & w_{c_2} & \cdots & w_{c_t} & \cdots & w_{c_n} \end{array} \right] \tag{15}$$

where $\forall c_t, 1 \leq t \leq n, w_{c_t}$ is a pair (τ_{c_t}, ϕ_{c_t}) representing whether the corresponding process node is a terminal node, a connect node, or neither.

From Equations 9 and 13, we define a row weight vector as follows:

$$W^r_{iter} = \left[\begin{array}{ccccccc} w_{r_1} & w_{r_2} & \cdots & w_{r_s} & \cdots & w_{r_m} \end{array} \right]^T \tag{16}$$

where m is the number of resources, and $\forall r_s, 1 \leq s \leq m, w_{r_s}$ is a pair (τ_{r_s}, ϕ_{r_s}) representing whether the corresponding resource node is a terminal node, a connect node, or neither.

3.3.2 DDU Operation Examples

We now illustrate a series of logical operations of the DDU with two simple examples. In one example, the current state of a system consisting of two processes and three resources has a deadlock. In the second example, the current state of the same system does not have a deadlock.

Example 18 Two Processes and Three Resources with a Cycle

An SoC shown in Figure 17 has two processes, p_1 running on DSP and p_2 running on VSP, and three resources, ImC, PCI and WI as q_1, q_2 and q_3, respectively. The matrix representation of this example is shown in Figure 18.

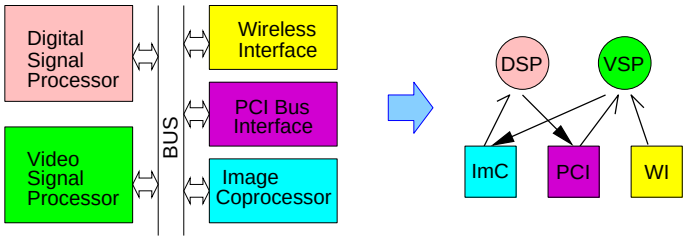


Figure 17: SoC example with two processors and three resources.

P \ Q	p_1(DSP)	p_2(VSP)
q_1(ImC)	g	r
q_2(PCI)	r	g
q_3(WI)	0	g

Figure 18: SoC example with two processors and three resources with a cycle.

We now show how the parallel operation of the DDU works cycle by cycle. In the beginning, given γ_{ij} shown informally in Figure 17(b), lines 2-6 of Algorithm 2 construct an initial matrix M_{ij} shown in Equation 17. Each element in M_{ij} is referred to as α_{st}, where $1 \leq s \leq m$ and $1 \leq t \leq n$.

Here, each α_{st} can have a value of either g, r or 0.

$$M_{ij} = \begin{bmatrix} \alpha_{11} & \alpha_{12} \\ \alpha_{21} & \alpha_{22} \\ \alpha_{31} & \alpha_{32} \end{bmatrix} = \begin{bmatrix} g & r \\ r & g \\ 0 & g \end{bmatrix} = M_{iter_1} \text{ (at the first iteration)} \tag{17}$$

Then, Line 7 of Algorithm 2 calls Algorithm 1. However, instead of the call, M_{ij} just becomes M_{iter_1} by a detection start signal (as stated in Section 3.3.1). Since M_{iter_1} needs to be processed in a two-dimensional operation, we represent M_{iter_1} as two-bit binary encoding column and row representations, $M^c_{iter_1}$ and $M^r_{iter_1}$, respectively, which are shown in Equation 18. Again, note that these two matrices are the same as M_{iter_1} and they are only shown for the purpose of the understanding of the parallel two-dimensional operation inside the DDU.

$$M^c_{iter_1} = \begin{bmatrix} 01 & 10 \\ 10 & 01 \\ 00 & 01 \end{bmatrix}, \; M^r_{iter_1} = \left[\begin{array}{cc} 0 & 1 \\ 1 & 0 \\ \hline 1 & 0 \\ 0 & 1 \\ \hline 0 & 0 \\ 0 & 1 \end{array}\right] \tag{18}$$

Now a Bit-Wise-Or ($BWO, \vee$) operation is applied to each column of $M^c_{iter_1}$, resulting in the $BWO^c_{iter_1}$ matrix shown in Equation 19. Similarly, another BWO operation is applied to each row of $M^r_{iter_1}$ resulting in the $BWO^r_{iter_1}$ matrix shown in Equation 19. These BWO operations are processed in parallel and the results are fed to the next operation.

$$BWO^c_{iter_1} = \begin{bmatrix} 11 & 11 \end{bmatrix}, \; BWO^r_{iter_1} = \left[\begin{array}{c} 1 \\ 1 \\ \hline 1 \\ 1 \\ \hline 0 \\ 1 \end{array}\right] \tag{19}$$

Next, an eXclusive-OR (XOR, $\oplus$) operation is applied to both bits of each entry of $BWO^c_{iter_1}$ resulting in the $XOR^c_{iter_1}$ matrix shown in Equation 20. For instance, if $BWO^c = \begin{bmatrix} 11 & 01 \end{bmatrix}$, then XOR^c will be $\begin{bmatrix} 1 \oplus 1 & 0 \oplus 1 \end{bmatrix} = \begin{bmatrix} 0 & 1 \end{bmatrix}$. A similar XOR operation is also applied to both bits of each entry of $BWO^r_{iter_1}$ resulting in the $XOR^r_{iter_1}$ matrix shown in Equation 20.

$$XOR^c_{iter_1} = \begin{bmatrix} \tau_{c_1} & \tau_{c_2} \end{bmatrix} = \begin{bmatrix} 0 & 0 \end{bmatrix}, \; XOR^r_{iter_1} = \begin{bmatrix} \tau_{r_1} & \tau_{r_2} & \tau_{r3} \end{bmatrix}^T = \begin{bmatrix} 0 & 0 & 1 \end{bmatrix}^T \tag{20}$$

An element '1' in $XOR^c_{iter_1}$ represents that the corresponding column (recall a column corresponds to a process) in M_{iter_1} is a terminal process node; thus, edge(s) in the column are reducible.

On the other hand, an element '1' in $XOR^r_{iter_1}$ indicates that the corresponding row (recall a row corresponds to a resource) in M_{iter_1} is a terminal resource node; hence, edge(s) in the row are reducible. Since the element of the third row in $XOR^r_{iter_1}$ is '1,' which signifies the third row is a terminal row in matrix M_{iter_1}, the third row can be excluded from further consideration. That is, more iterations need to continue, which is shown in Equation 21.

$$T_{iter_1} \quad = \tau_C \vee \tau_R \quad = 1 \tag{21}$$

where $\tau_C = \bigvee_{t=1}^{n} \tau_{c_t} = 0$ and $\tau_R = \bigvee_{s=1}^{m} \tau_{r_s} = 1$. Since T_{iter_1} in Equation 21 results in '1,' there exists at least one terminal edge in this iteration; thus, further iteration(s) are necessary. Before continuing the next iteration, we calculate connect nodes, $AND^c_{iter_1}$ and $AND^r_{iter_1}$, shown in Equation 22.

$$AND^c_{iter_1} = \begin{bmatrix} \phi_{c_1} & \phi_{c_2} \end{bmatrix} = \begin{bmatrix} 1 & 1 \end{bmatrix},\ AND^r_{iter_1} = \begin{bmatrix} \phi_{r_1} & \phi_{r_2} & \phi_{r3} \end{bmatrix}^T = \begin{bmatrix} 1 & 1 & 0 \end{bmatrix}^T \tag{22}$$

Weight vectors are $W^c_{iter_1}$ and $W^r_{iter_1}$, shown in Equation 23.

$$\begin{aligned} W^c_{iter_1} &= \begin{bmatrix} w_{c_1} & w_{c_2} \end{bmatrix} = \begin{bmatrix} (0,1) & (0,1) \end{bmatrix}, \\ W^r_{iter_1} &= \begin{bmatrix} w_{r_1} & w_{r_2} & w_{r3} \end{bmatrix}^T = \begin{bmatrix} (0,1) & (0,1) & (1,0) \end{bmatrix}^T \end{aligned} \tag{23}$$

where (0,1) signifies a connect node and (1,0) signifies a terminal node. Thus, row 3 is a terminal node. After the terminal edge revealed in the first iteration is eliminated by Equation 11, the next iteration begins with new M_{iter_2} shown in Equation 24.

$$M_{iter_2} = \begin{bmatrix} \alpha_{11} & \alpha_{12} \\ \alpha_{21} & \alpha_{22} \\ \alpha_{31} & \alpha_{32} \end{bmatrix} = \begin{bmatrix} g & r \\ r & g \\ 0 & 0 \end{bmatrix} \text{ (at the second iteration)} \tag{24}$$

$M^c_{iter_2}$ and $M^r_{iter_2}$ at iteration 2 are shown in Equation 25.

$$M^c_{iter_2} = \begin{bmatrix} 01 & 10 \\ 10 & 01 \\ 00 & 00 \end{bmatrix},\ M^r_{iter_2} = \left[\begin{array}{cc} 0 & 1 \\ 1 & 0 \\ \hline 1 & 0 \\ 0 & 1 \\ \hline 0 & 0 \\ 0 & 0 \end{array}\right] \tag{25}$$

Then, $BWO^c_{iter_2}$ and $BWO^r_{iter_2}$ are shown in Equation 26.

$$BWO^c_{iter_2} = \begin{bmatrix} 11 & 11 \end{bmatrix},\ BWO^r_{iter_2} = \left[\begin{array}{c} 1 \\ 1 \\ \hline 1 \\ 1 \\ \hline 0 \\ 0 \end{array}\right] \tag{26}$$

Next, Equation 27 shows $XOR^c_{iter_2}$ and $XOR^r_{iter_2}$, which represent that no terminal nodes exist in M_{iter_2}.

$$XOR^c_{iter_2} = \begin{bmatrix} 0 & 0 \end{bmatrix},\ XOR^r_{iter_2} = \begin{bmatrix} \frac{0}{0} \\ \hline 0 \end{bmatrix} \tag{27}$$

As a result, T_{iter_2} becomes '0' as shown in Equation 28, which means M_{iter_2} is irreducible.

$$T_{iter_2} = \tau_C \vee \tau_R = 0 \tag{28}$$

since $\tau_C = \bigvee_{t=1}^{n} \tau_{c_t} = 0$ and $\tau_R = \bigvee_{s=1}^{m} \tau_{r_s} = 0$. Therefore, this iteration is the last. At this moment, we need to find D_{iter_2}, the deadlock decision result. To do this, we first calculate $AND^c_{iter_2}$ and $AND^r_{iter_2}$, as shown in Equation 29, which represent the existence of connect nodes in columns and in rows, respectively.

$$AND^c_{iter_2} = \begin{bmatrix} 1 & 1 \end{bmatrix} \text{ and } AND^r_{iter_2} = \begin{bmatrix} 1 & 1 & 0 \end{bmatrix}^T \tag{29}$$

Weight vectors at this iteration are $W^c_{iter_2}$ and $W^r_{iter_2}$, shown in Equation 30.

$$W^c_{iter_2} = \begin{bmatrix} (0,1) & (0,1) \end{bmatrix},\ W^r_{iter_2} = \begin{bmatrix} (0,1) & (0,1) & (0,0) \end{bmatrix}^T \tag{30}$$

where (0,1) signifies a connect node and (0,0) in $W^r_{iter_2}$ signifies no edges in the third row. Next, the decision of a deadlock is made by Equation 31.

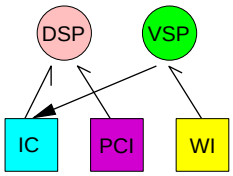
$$D_{iter_2} = \phi_c \vee \phi_r = 1, \qquad \text{when } T_{iter_2} = 0 \tag{31}$$

where $\phi_c = \bigvee_{t=1}^{n} \phi_{c_t} = 1$ and $\phi_r = \bigvee_{s=1}^{m} \phi_{r_s} = 1$. Since (i) $T_{iter_2} = 0$, which signifies that this new $M_{i,j+1}$ is not reducible any more, and (ii) $D_{iter_2} = 1$, which signifies that edges still exist, we finally conclude that γ_{ij} has a deadlock. ■

Example 19 Two Processes and Three Resources without a Cycle

Consider the same system as shown in Example 18. However, the system currently has a different set of request and grant edges as shown in Figure 19.

We now show how the parallel operation of the DDU works cycle by cycle in this case, which has no cycles. First, given γ_{ij} shown in Figure 19(a), initial matrix M_{ij} shown below is constructed



(a) a RAG

P \ Q	p_1(DSP)	p_2(VSP)
q_1(ImC)	g	r
q_2(PCI)	g	0
q_3(WI)	0	g

(b) M_{ij}

Figure 19: SoC example with two processors and three resources without a cycle.

by lines 2-6 of Algorithm 2.

$$M_{ij} = \begin{bmatrix} \alpha_{11} & \alpha_{12} \\ \alpha_{21} & \alpha_{22} \\ \alpha_{31} & \alpha_{32} \end{bmatrix} = \begin{bmatrix} g & r \\ g & 0 \\ 0 & g \end{bmatrix} = M_{iter_1} \text{ (at the first iteration)} \tag{32}$$

Second, two corresponding binary coded matrices $M^c_{iter_1}$ and $M^r_{iter_1}$, shown in Equation 33, are constructed according to the binary encoding scheme, explained in Section 3.3.1.

$$M^c_{iter_1} = \begin{bmatrix} 01 & 10 \\ 01 & 00 \\ 00 & 01 \end{bmatrix}, \; M^r_{iter_1} = \left[\begin{array}{cc} 0 & 1 \\ 1 & 0 \\ \hline 0 & 0 \\ 1 & 0 \\ \hline 0 & 0 \\ 0 & 1 \end{array}\right] \tag{33}$$

Third, a Bit-Wise-Or operation is applied to each column of $M^c_{iter_1}$, resulting in the $BWO^c_{iter_1}$ matrix shown in Equation 34. At the same time, a similar Bit-Wise-Or operation is applied to each row of $M^r_{iter_1}$, resulting in the $BWO^r_{iter_1}$ matrix.

$$BWO^c_{iter_1} = \begin{bmatrix} 01 & 11 \end{bmatrix}, \; BWO^r_{iter_1} = \left[\begin{array}{c} 1 \\ 1 \\ \hline 0 \\ 1 \\ \hline 0 \\ 1 \end{array}\right] \tag{34}$$

Fourth, an eXclusive-OR($\oplus$) operation is applied to the two bits of each entry of $BWO^c_{iter_1}$, resulting in the $XOR^c_{iter_1}$ matrix shown in Equation 35. Likewise, another eXclusive-OR operation is also applied to the two bits of each entry of $BWO^r_{iter_1}$, resulting in the $XOR^r_{iter_1}$ matrix.

$$XOR^c_{iter_1} = \begin{bmatrix} 1 & 0 \end{bmatrix}, \; XOR^r_{iter_1} = \left[\begin{array}{c} 0 \\ \hline 1 \\ \hline 1 \end{array}\right] \tag{35}$$

Now, the termination condition is calculated by Equation 36.

$$T_{iter_1} = \tau_C \vee \tau_R = 1 \tag{36}$$

where $\tau_C = \bigvee_{t=1}^{n} \tau_{c_t} = 1$ and $\tau_R = \bigvee_{s=1}^{m} \tau_{r_s} = 1$. Since T_{iter_1} in Equation 36 results in '1,' there exists at least one more terminal edge at this iteration; thus, at least one more iteration is necessary.

In Equation 35, since the element of the first column in $XOR^c_{iter_1}$ is '1,' which means the first column is a terminal column in matrix M_{iter_1}, the first column can be eliminated from further consideration. Also, since the elements of second and third rows in $XOR^r_{iter_1}$ are '1,' which means the second and third rows are terminal rows in matrix M_{iter_1}, the second and third rows can be eliminated from further consideration. After the terminal edges revealed in iteration one are eliminated using Equation 11, the next iteration begins with new M_{iter_2}, shown in Equation 37.

$$M_{iter_2} = \begin{bmatrix} \alpha_{11} & \alpha_{12} \\ \alpha_{21} & \alpha_{22} \\ \alpha_{31} & \alpha_{32} \end{bmatrix} = \begin{bmatrix} 0 & r \\ 0 & 0 \\ 0 & 0 \end{bmatrix} = \begin{bmatrix} 00 & 10 \\ 00 & 00 \\ 00 & 00 \end{bmatrix} \tag{37}$$

$BWO^c_{iter_2}$, $XOR^c_{iter_2}$, $BWO^r_{iter_2}$ and $XOR^r_{iter_2}$ are shown in Equations 38 and 39, respectively.

$$BWO^c_{iter_2} = \begin{bmatrix} 00 & 10 \end{bmatrix},\ BWO^r_{iter_2} = \begin{bmatrix} 1 \\ 0 \\ \hline 0 \\ 0 \\ \hline 0 \\ 0 \end{bmatrix} \tag{38}$$

$$XOR^c_{iter_2} = \begin{bmatrix} 0 & 1 \end{bmatrix},\ XOR^r_{iter_2} = \begin{bmatrix} 1 \\ \hline 0 \\ \hline 0 \end{bmatrix} \tag{39}$$

The termination condition is shown in Equation 40.

$$T_{iter_2} = \tau_C \vee \tau_R = 1 \tag{40}$$

where $\tau_C = \bigvee_{t=1}^{n} \tau_{c_t} = 1$ and $\tau_R = \bigvee_{s=1}^{m} \tau_{r_s} = 1$. After the terminal edge revealed at the second iteration is eliminated by Equation 11, the next iteration begins with new M_{iter3}, shown in Equation 41.

$$M_{iter3} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix} \tag{41}$$

Since this M_{iter3} has no edges, all the remaining results of the deadlock equations will turn out to be '0'; thus, the DDU identifies that the system state γ_{ij} does not have a deadlock. ■

3.3.3 On the Relationship between the DDU and Two Level Logic Minimization

We here briefly mention the similarities between the concept of terminal reduction and two-level logic minimization, the general concept of which is described in Chapter 2 of [10]. The Quine-McCluskey (QM) method used to find minimum prime implicants (covering a given function) iteratively reduces "a set of standard sum of products form" to "a set of prime implicants" by using the concepts of dominance and essentiality [32]. By contrast, Coudert et al. propose an efficient algorithm that directly computes a set of essential elements using Binary Decision Diagrams (BDDs) and metaproducts, which makes its complexity independent from the numbers of minterms and prime implicants of a function (instead of computing the dominance relations, whose huge BDDs are the bottleneck of previous algorithms) [9]. Our approach of parallel terminal reduction based on a matrix is very similar to the QM method in terms of the representation of the design space by use of a matrix and in that both methods remove columns and/or rows to come up with their solutions; however, matrix row and column removal criteria in each differ according to their purposes. While Quine-McCluskey uses a matrix of minterms and prime implicants, our method uses a matrix of resources and requesters. Another difference is that while traditional logic minimization algorithms typically use *don't cares*, each element of a matrix of our method has a ternary value, defined in Definition 34, with no equivalent *don't care* concept in our case. A big difference in terms of run-time complexity lies in that while our approach is linear, exact (global minimum) logic minimization algorithms (such as QM) are either polynomial or exponential.

3.3.4 Detailed Description of the DDU Architecture

This subsection describes the architecture of the DDU. The DDU consists of three parts as shown in Figure 8: matrix cells (part 1), weight cells (part 2) and a decide cell (part 3). Part 1 of the DDU represents the system state matrix M_{ij} via use of an array of matrix cells that represents an array of $(\alpha^r_{st}, \alpha^g_{st})$ entries where $1 \leq s \leq m$ and $1 \leq t \leq n$. Part 2

consists of two weight vectors: (i) a column weight vector W^c (shown in Equation 15) below the matrix cells and (ii) a row weight vector W^r (shown in Equation 16) on the right-hand side of the array of matrix cells. Each element w_{c_t}, $1 \leq t \leq n$, in W^c is called a column weight cell, and each element w_{r_s}, $1 \leq s \leq m$, in W^r is called a row weight cell. At the bottom right corner of the DDU is one decide cell (part 3) which calculates T_{iter} in Equation 10 and D_{iter} in Equation 14 in order to check deadlock. All cells are interconnected appropriately via buses.

Figure 8, repeated here as Figure 20 for convenience, illustrates the architecture of the DDU for three processes and three resources. This DDU has nine matrix cells (3×3) for each edge element $(\alpha^r_{st}, \alpha^g_{st})$ of M_{ij}, six weight cells (three for column processing and three for row processing), and one decide cell for deciding whether or not deadlock has been detected.

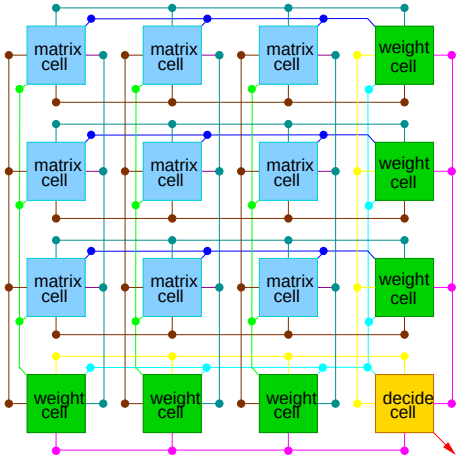


Figure 20: DDU architecture.

Figure 21 shows a matrix cell which corresponds to an entry $\alpha_{st} = (\alpha^r_{st}, \alpha^g_{st})$. The matrix cell has one 2-bit register (which stores the edge information $(\alpha^r_{st}, \alpha^g_{st})$) with two input and four output signals. One pair of outputs (C^r_{st}, C^g_{st}) goes to a column weight cell w_{c_t} while the other pair of outputs (R^r_{st}, R^g_{st}) goes to a row weight cell w_{r_s}. Please note that C^r_{st} and R^r_{st} are the same, and C^g_{st} and R^g_{st} are the same. Two inputs (τ_{r_s} and τ_{c_t}) from

weight cells are used to suppress outputs to reflect the terminal reduction when this cell belongs to a terminal node (i.e., when this matrix cell corresponds to a terminal edge, this cell must be excluded from further iterations).

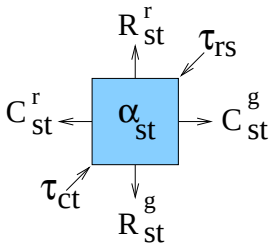


Figure 21: Matrix Cell α_{st} in matrix array M_{ij}.

Figure 22 depicts a logic diagram of a matrix cell for row one and column one of Example 18. All outputs are controlled by the OR of τ_{c_1} and τ_{r_1}, which indicates whether or not the corresponding column or row is a terminal node.

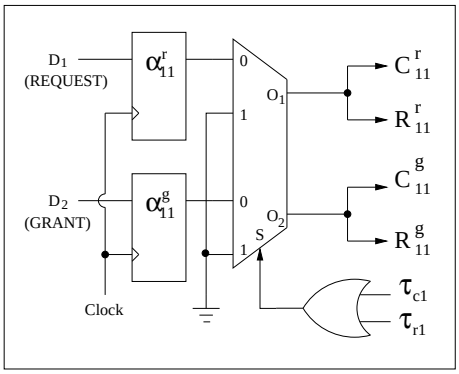


Figure 22: Logic diagram of a matrix cell.

Example 20 Operation of a matrix cell

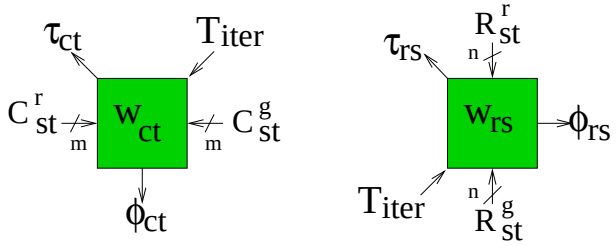
Consider the matrix cell shown in Figure 22, which corresponds to a cell $\alpha_{11} = (\alpha^r_{11}, \alpha^g_{11})$ of Equation 17. Since the cell has a grant edge, currently $\alpha_{11} = (0, 1)$, which has been latched by the input data $D_1 = 0$ and $D_2 = 1$. Hence, outputs will be $(C^r_{11}, C^g_{11}) = (R^r_{11}, R^g_{11}) = (0, 1)$ if both τ_{c_1} and τ_{r_1} are false (i.e., this cell belongs to neither a terminal column nor a terminal row). If, however, this cell belongs to either a terminal column (i.e, $\tau_{c_1} = 1$) or a terminal row (i.e., $\tau_{r_1} = 1$), the output of the OR-gate connected to the S (select) input of the MUX in Figure 22 is true; thus, the MUX will

select input-1's, which are grounded; therefore, all four outputs of the matrix cell will be zero. This latter process is called "suppression" as stated in Section 3.3 and emulates edge reduction. ■

A column weight cell w_{c_t} shown in Figure 23(a) has *2m+1* inputs (where m is the number of resources) and generates two outputs. All m pairs of inputs $(C^r_{st}, C^g_{st}), 1 \leq s \leq m$, of w_{c_t} come from matrix cells in column c_t. T_{iter} is an input from a decide cell. Outputs are a pair (τ_{c_t}, ϕ_{c_t}). All column weight cells calculate BWO^c_{iter} in Equation 6, XOR^c_{iter} in Equation 8, and AND^c_{iter} in Equation 12; the result of this calculation reveals all terminal process nodes in the current iteration. Each column weight cell finally produces (τ_{c_t}, ϕ_{c_t}), representing whether the corresponding process node is a terminal node, a connect node, or neither as shown in Equation 23.

Each weight cell has two registers that store the result of each iteration, and T_{iter} is used to stop iterations.

A row weight cell w_{r_s} shown in Figure 23(b) has *2n+1* inputs (where n is the number of processes) and generates two outputs. All n pairs of inputs $(R^r_{st}, R^g_{st}), 1 \leq t \leq n$, come from matrix cells in row r_s. Outputs are a pair (τ_{r_s}, ϕ_{r_s}). All row weight cells calculate BWO^r_{iter} in Equation 7, XOR^r_{iter} in Equation 9, and AND^r_{iter} in Equation 13; the result of this calculation reveals all terminal resource nodes. Each row weight cell finally produces (τ_{r_s}, ϕ_{r_s}), representing whether the corresponding resource node is a terminal node, a connect node, or neither as shown in Equation 23.



(a) a column weight cell (b) a row weight cell

Figure 23: Weight Cells w_{c_t} and w_{r_s} described in Equations 15 and 16.

Figure 24 depicts a logic diagram of the weight cell for the first column of Example 18. The weight cell for the second column of Example 18 would be exactly the same but with inputs C_{12}^r, C_{22}^r and C_{32}^r to the top OR gate, inputs C_{12}^g, C_{22}^g and C_{32}^g to the bottom OR gate, and a pair of outputs (τ_{c_2}, ϕ_{c_2}).

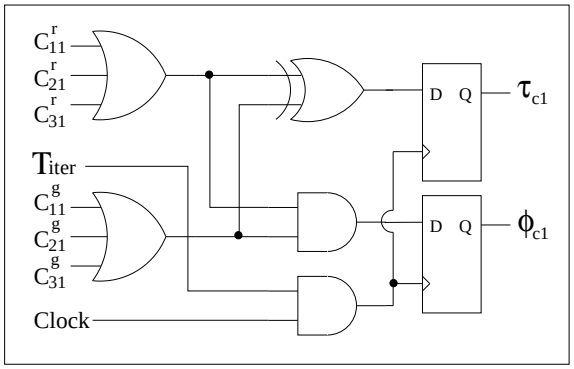


Figure 24: Logic diagram of a column weight cell.

Example 21 Operation of a weight cell

Consider the weight cell shown in Figure 24, which corresponds to the weight cell for the first column of Example 18. We assume that the DDU is currently at the first iteration of the terminal reduction sequence. At this moment, $(C_{11}^r, C_{11}^g) = (0,1)$, $(C_{21}^r, C_{21}^g) = (1,0)$ and $(C_{31}^r, C_{31}^g) = (0,0)$, as shown in the first column of $M_{iter_1}^c$ of Equation 18 in Example 18. Thus, outputs w_{c_1} i.e., (τ_{c_1}, ϕ_{c_1}) will be $(((C_{11}^r \vee C_{21}^r \vee C_{31}^r) \oplus (C_{11}^g \vee C_{21}^g \vee C_{31}^g)), ((C_{11}^r \vee C_{21}^r \vee C_{31}^r)$ AND $(C_{11}^g \vee C_{21}^g \vee C_{31}^g))) = (0,1)$ as shown in Equation 23 of Example 18. ■

The decide cell shown in Figure 25 has $2m + 2n$ inputs and two outputs. All m pairs of inputs $(\tau_{r_s}, \phi_{r_s}), 1 \leq s \leq m$, are connected to their corresponding row weight cells, while all n pairs of inputs $(\tau_{c_t}, \phi_{c_t}), 1 \leq t \leq n$, are connected to their corresponding column weight cells. The two outputs are T_{iter} (shown in Equation 10), which indicates whether or not M_{iter} is reducible further, and D_{iter} (shown in Equation 14), which, when $T_{iter} = 0$, indicates whether or not the system state has a deadlock.

Figure 26 depicts a logic diagram of the decide cell for Example 18. The three inputs ϕ_{r_1}, ϕ_{r_2} and ϕ_{r3} come from three row weight cells, and two inputs ϕ_{c_1} and ϕ_{c_2} come from

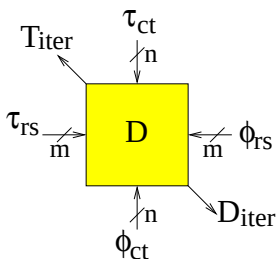


Figure 25: Decide cell D.

two column weight cells. Similarly, the three inputs τ_{r_1}, τ_{r_2} and τ_{r3} come from three row weight cells, and two inputs τ_{c_1} and τ_{c_2} come from two column weight cells. One output D_{iter}, which indicates the existence of connect nodes, is the result of deadlock detection, and the other output T_{iter} indicates the existence of terminal edges.

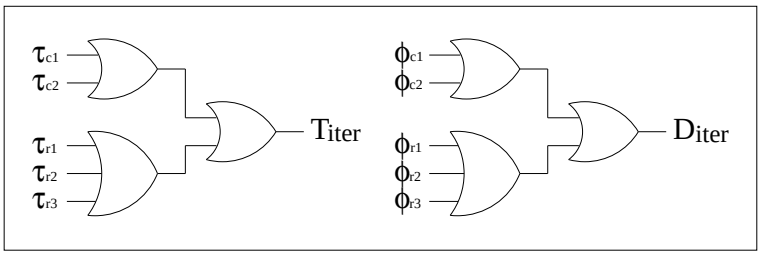


Figure 26: Logic diagram of a decide cell.

Example 22 Operation of a decide cell

Consider the decide cell shown in Figure 26, which corresponds to the decide cell of Example 18. We assume that the DDU is currently at the first iteration of the terminal reduction sequence. The decide cell inputs which determine output T_{iter} are $XOR^c_{iter_1}$ and $XOR^r_{iter_1}$ as shown in Equation 20 of Example 18. Thus, T_{iter} will also be true (implying further iterations are necessary), because $T_{iter} = \tau_C \vee \tau_R = (\tau_{c_1} \vee \tau_{c_2}) \vee (\tau_{r_1} \vee \tau_{r_2} \vee \tau_{r_3}) = (0 \vee 0) \vee (0 \vee 0 \vee 1) = 0 \vee 1$ is true, as shown in Equation 21. Likewise, the decide cell inputs which determine output D_{iter} are $AND^c_{iter_1}$ and $AND^r_{iter_1}$ as shown in Equation 22. Thus, $D_{iter} = \phi_C \vee \phi_R = (\phi_{c_1} \vee \phi_{c_2}) \vee (\phi_{r_1} \vee \phi_{r_2} \vee \phi_{r_3}) = (1 \vee 1) \vee (1 \vee 1 \vee 0) = 1 \vee 1$ is true; however, this output is meaningless until T_{iter} becomes false. ■

In this section, we explained the circuitry and operation of the DDU with examples in detail. In the next section, we will show the synthesis result of various DDU sizes.

3.3.5 Synthesis Result of the DDU

We implement the DDU in Verilog using a mixture of Register Transfer Level (RTL) and behavioral level code. We use the Synopsys Design Compiler to synthesize the DDU with a $0.3\mu m$ standard cell library from AMIS [5]. Table 1 shows the synthesis results of five types of DDUs customized according to the number of processes and resources in an SoC. The fourth column, denoted "logic delay per iteration," represents a logic delay per each iteration for the corresponding DDU. The fifth column, denoted "worst case # iterations," represents the number of worst case iterations for the corresponding DDU. The sixth column, denoted "worst case delay," results from "logic delay per one iteration" multiplied by "worst case # iterations." Table 1 reveals the following. (i) The worst case number of iterations increases linearly with the smaller number out of the number of processes and the number of resources. (ii) The logic delay per iteration increases proportional to the larger number out of the number of processes and the number of resources. (iii) The other numbers in Table 1 increase almost quadratically proportional to the total number of processes plus resources.

Table 1: Synthesized result of the DDU.

# processes × # resources	lines of Verilog	area in terms of two-input NAND gates	logic delay per iteration (ns)	worst case # iterations	the worst case delay (ns)
2×3	49	186	0.91	2	1.82
5×5	73	364	2.21	6	13.26
7×7	102	455	2.51	10	25.1
10×10	162	622	3.66	16	58.56
50×50	2682	14142	4.12	96	395.52

3.4 Experiments

3.4.1 Experimental Setup

In the experiments, we implement in Verilog HDL the MPSoC architecture shown in Figure 5(a) (repeated here as Figure 27) except for the PE cores; for the PowerPC cores in

Figure 27, we use an Instruction Set Simulator (ISS) provided by simulation tool vendor Mentor Graphics – specifically, the ISS is provided as a "processor support package" for the use in the Seamless Co-Verification Environment (CVE) [33]. The MPSoC has four Motorola MPC755s as processing elements (PEs). The MPC755 has two separate instruction and data L1 caches each of size 32KB. The MPSoC has also four resources: a Video Interface (VI) device, a Fast Fourier Transform (FFT) unit, an Inverse Discrete Cosine Transform (IDCT) unit and a Wireless Interface (WI) device. These four resources have timers, interrupt generators and input/output ports that are necessary to support our simulations. In addition, the MPSoC has a DDU for five processes and five resources, an arbiter and 16MB of shared memory. The master clock rate of the bus system is 10 ns (100 MHz). Code for each MPC755 runs on an instruction-accurate (not cycle-accurate) MPC755 simulator provided by Seamless CVE.

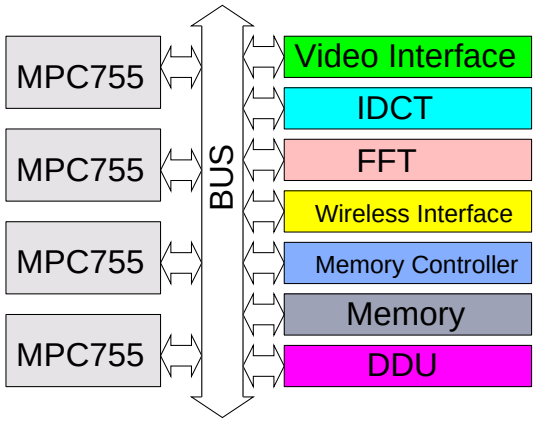


Figure 27: MPSoC architecture for the DDU evaluation.

We assume the following in our experiments. (i) The MPSoC is capable of capturing still and motion pictures, processing IDCT, and performing signal and image processing. (ii) The MPSoC can also support data streaming applications using a standard wireless LAN card. (iii) Such functionalities described in (i) are implemented partly in hardware in this MPSoC; thus, each PE will likely request the services of some of the hardware units.

On top of the MPSoC, we use Atalanta RTOS version 0.3 [51] for multiprocessing

introduced in Section 1.4.2. The RTOS code resides in the shared memory, and all PEs execute the same RTOS code and share kernel structures as well as states of all processes and resources.

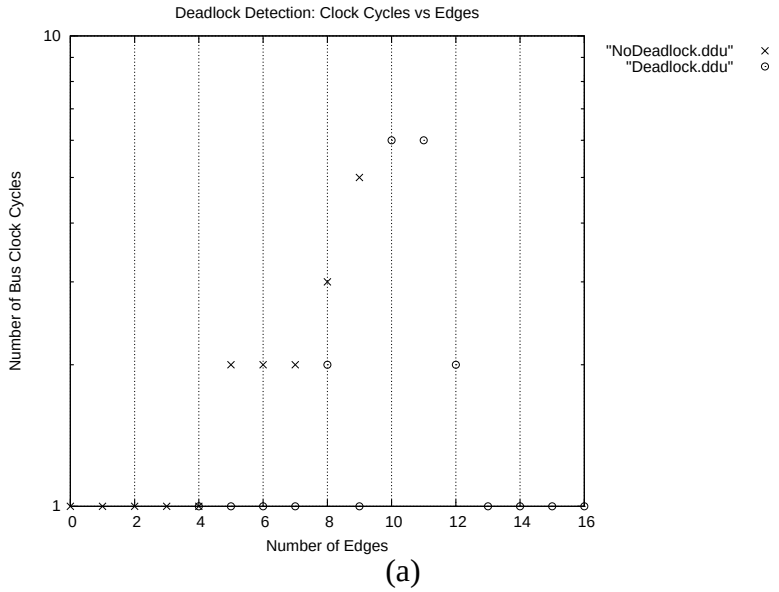
With the MPSoC and the Atalanta RTOS, we completed two experiments with three deadlock detection implementations: (i) the DDU, (ii) a software implementation of PDDA and (iii) a software implementation of an $O(m \times n)$ deadlock detection algorithm[3]. One was a performance comparison among the three implementations while the other experiment was a comparison of the execution time of an application using DDU hardware (i) versus using the faster software deadlock detection algorithm (ii). We accomplished the two experiments through instruction-accurate simulations. The simulations were carried out using Mentor Graphics Seamless CVE [33], aided by Synopsys VCS [53] for Verilog HDL simulation and XRAY [34] for software debugging.

For the experimentation, we implemented three versions of PDDA according to the numbers of resources and processes. One PDDA implementation was for a system having four resources and four processes, another implementation was for five resources and five processes, and the third PDDA implementation was for eight resources and eight processes; thus, PDDA code size was fitted to the maximum expected numbers of processes and resources.

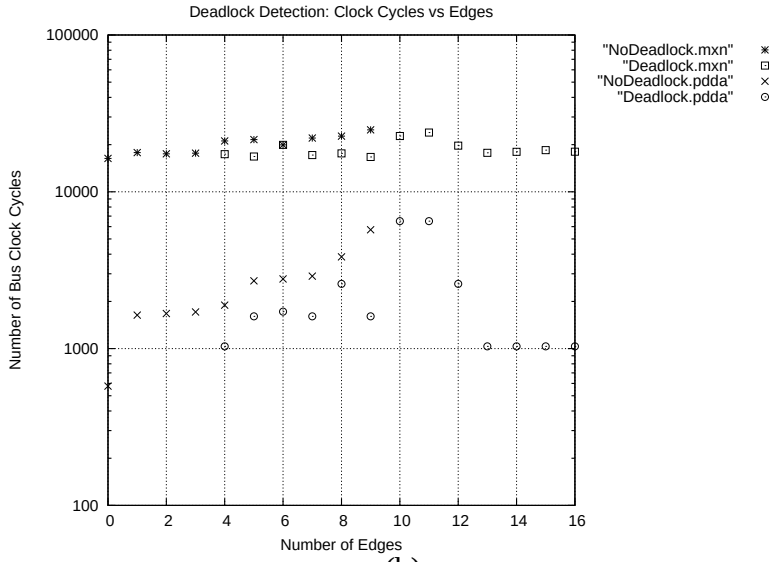
3.4.2 Execution Time Comparison of PDDA

In the first experiment, with the MPSoC described in Section 3.4.1, we have simulated a large number of deadlock detection cases with various numbers of request and grant edges (arbitrarily chosen) to measure the execution time difference among the three implementations. The cases are ten non-deadlocked sets of request and grant edges with the number of edges spanning from 0 to 9, and thirteen deadlocked sets of request and grant edges with the number of edges spanning from 4 to 16, as shown in Figure 28.

[3] Algorithm 4.4 in [31]



(a)



(b)

Figure 28: Run-time comparison of deadlock detection algorithms.

All simulations were executed in a system having five resources and five processes. In the legend of Figure 28, "NoDeadlock.ddu," "NoDeadlock.pdda" and "NoDeadlock.mxn" represent non-deadlocked scenarios checked by the DDU, PDDA in software and an $O(m \times n)$ deadlock detection algorithm[4] in software, respectively. By contrast, "Deadlock.ddu,"

[4] Algorithm 4.4 in [31]

"Deadlock.pdda" and "Deadlock.mxn" represent deadlocked scenarios checked by the three implementations, respectively. Please note that even though our results demonstrate three orders of magnitude speedups as compared to other software algorithms (including PDDA in software) as shown in Figure 28, since the result could be off as much as one order of magnitude due to instruction accurate (not cycle accurate) simulations for the MPC755, we only claim two orders of magnitude speedups in the following. In summary, Figure 28 demonstrates, in all cases, (i) two orders of magnitude or greater difference in the number of cycles between the DDU and the other software implementations, and (ii) about one order of magnitude difference in average between PDDA and the $O(m \times n)$ software algorithm. The reason for the latter is that while PDDA in software does bit-wise operations, the $O(m \times n)$ software algorithm executes many extra instructions to traverse nodes, search linked lists and update data structures. Please note that this comparison is only a part of a bigger picture, which we will show in the next experiment.

3.4.3 Execution Time Comparison of an Application

In the second experiment, we wanted to identify the difference in an application executing using the DDU versus PDDA in software (note that our software version of PDDA executes much faster than the $O(m \times n)$ deadlock detection algorithm[5] in software as shown in Figure 28). We devised an application example inspired by the Jini lookup service system [38] in which client applications can request services through intermediate layers (i.e., lookup, discovery and admission). Since the SoC, introduced in Section 3.4.1, has multiple processes and multiple resources, and because Conditions 1$\sim$5 in Section 1.3 could also potentially all be satisfied during the normal execution of the application, a deadlock is possible in such a system. Thus, this is a proper example of a practical application that can benefit from the DDU. In this experiment, we invoked one process on each PE and prioritized all processes, p_1 being the highest and p_4 being the lowest. The video frame we

[5] Algorithm 4.4 in [31]

use for the experiment is a test frame whose size is 64 by 64 pixels. The IDCT processing time of the test frame takes approximately 23,600 bus clock cycles.

We show a sequence of requests and grants that finally leads to a deadlock as shown in Table 2 and Figure 29. Process p_1, running on PE1, requests both the IDCT and the VI at time t_1, which are then granted to p_1. After that, p_1 starts receiving a video stream through the VI and does IDCT processing. At time t_2, process p_3, running on PE3, needs and requests the IDCT and the WI to simultaneously convert a frame to an image and send the image through the WI. However, only the WI is granted to p_3 since the IDCT is unavailable. At time t_3 shown in Table 2 and Figure 29, p_2 running on PE2 also requests the IDCT and WI hardware units, which are not available for p_2. When the IDCT is released by p_1 at time t_4, the IDCT is granted to p_2 since p_2 has a higher priority than p_3. This last grant will lead to a deadlock in the SoC.

Table 2: A sequence of requests and grants that leads to deadlock.

Time	Events No.	Events
t_0	e_0	The application starts.
t_1	e_1	p_1 requests IDCT and VI; IDCT and VI are granted to p_1 immediately.
t_2	e_2	p_3 requests IDCT and WI; WI is granted to p_3 immediately.
t_3	e_3	p_2 requests IDCT and WI. Both p_2 and p_3 wait for IDCT.
t_4	e_4	IDCT is released by p_1.
t_5	e_5	IDCT is granted to p_2. since p_2 has a higher priority than p_3.

With the above scenario, we measured both the deadlock detection time Δ and the application execution time from the application start (t_0) until the detection of a deadlock in two cases: using (i) the DDU and (ii) PDDA in software. Please note that the RTOS initialization time was excluded (i.e., the RTOS is assumed to be fully operational at time

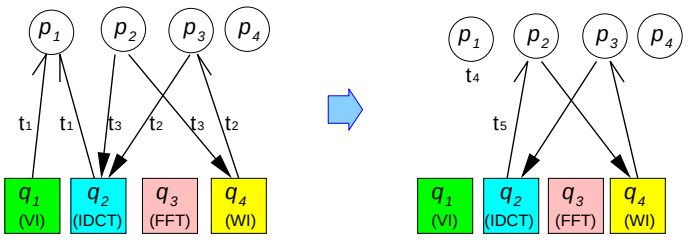


Figure 29: Events RAG for deadlock detection comparison.

t_0). Table 3 shows that (i) on average the DDU achieved a 1408X speedup over the software implementation of PDDA and that (ii) the DDU gave a 46% speedup of application execution time (i.e., from t_0 until deadlock detection after grant event e_5) over PDDA in software. The application invoked deadlock detection 10 times since deadlock detection is checked at every grant as well as request event. Please note that the algorithm run-time does not include the run-time of application programming interfaces. Please note also that in the above experimentation the application focuses on an initialization phase setting up a new set of tasks where interactions of resource requests and grants occur frequently. Thus, a different case where there exists no such dynamic resource usage and where deadlock does not occur so early would of course not show a 46% speedup, but instead would show a potentially far lower percentage speedup. Nonetheless, for critical situations where early deadlock detection is crucial, our approach can help significantly.

Table 3: Deadlock detection time and application execution time.

Method of Implementation	Algorithm Run Time*	Application Run Time*	Speedup[&]
DDU(hardware)	1.3	27714	$\frac{40523-27714}{27714} = 46\%$
PDDA in software	1830	40523	

*The time unit is a bus clock (10 ns in this example), and the values are averaged.
[&]The speedup is calculated according to the formula by Hennessy and Patterson [18].

3.5 Summary

This chapter describes a novel Parallel Deadlock Detection Algorithm called PDDA and illustrates a hardware implementation of PDDA called the Deadlock Detection Unit (DDU). This chapter provides their detailed descriptions, implementations, operation examples and synthesis results. Moreover, this chapter proves the correctness of PDDA and the run-time complexity of the DDU, which is $2 \times min(m, n) - 3 = O(min(m, n))$.

We carried out two sets of performance comparisons for the evaluation of this research. Both experiments were carried out through instruction-accurate simulations of a System-on-a-Chip (SoC) that consists of four Motorola MPC755s (each of which has 32KB separate instruction and data caches), a DDU, 16MB of shared memory and four types of resources.

The first experiment demonstrated that the DDU reduced the execution time of deadlock detection by 99% *(i.e., 100X)* or more as compared to two software implementations of deadlock detection algorithms. The second experiment showed that the DDU provided an application that exploits the DDU with a 46% speedup in execution time from the application start until the detection of a deadlock over the case in which the same application uses a software implementation of PDDA.

The two experimental results substantiate the following contributions. The DDU can provide developers with (i) a system that can check for deadlock more frequently (so far, deadlock detection in a real-time system has not been practical due to the long execution time of software deadlock detection that inevitably entails performance degradation); (ii) a system that has more design flexibility due to the time gained from using the DDU instead of software deadlock detection; and (iii) a system that has deadlock detection capability with a nearly zero performance penalty, which is significant because all deadlock detection methods proposed in previous work have much higher performance penalties [31].

In the next chapter, we will describe our novel deadlock avoidance approaches utilizing the DDU and present their performance evaluation.

CHAPTER IV

DEADLOCK AVOIDANCE UNIT

4.1 Introduction

Building on top of the proven properties of the DDU, it would be very helpful if there were a hardware unit that not only detects deadlock but also avoids potential deadlock within a few clock cycles and with a small amount of hardware.

In this chapter we present a new approach to deadlock avoidance, seeking to minimize the disadvantages (i)∼(v) mentioned in Section 2.2.4 such as prior declarations of maximum resource usage for each process (since such declarations may unfortunately not be practical or even possible in a real system, especially if the application code is updated often). Our novel approach mixes deadlock detection and avoidance (thus, not requiring advanced, a priori knowledge of resource requirements), contributing to easier adaptation of deadlock avoidance in an MPSoC by accommodating maximum freedom (i.e., maximum concurrency of requests and grants depending on a particular execution trace) with the advantage of deadlock avoidance. In other words, our new deadlock hardware solution, i.e., the Deadlock Avoidance Unit (DAU), requires neither prior knowledge about requirements of processes nor constraints of resource usage, yet achieves real-time deadlock avoidance; this constitutes the major novelty in our solution to deadlock avoidance.

The DAU, if employed, tracks all requests and releases of resources. In other words, the DAU receives, interprets and executes commands from processes; then the DAU returns command results back to processes. The DAU avoids deadlock by not allowing any grant or request that leads to a deadlock. In case of livelock resulting from an attempt to avoid request deadlock, the DAU asks one of the processes involved in the livelock to release resource(s) so that the livelock can also be resolved (for which we show Algorithm 5 in

Section 4.2.1). Likewise, in case of priority inversion resulting from an attempt to avoid grant deadlock (e.g., in Example 4 in Section 1.3, the grant deadlock could be avoided by granting q_2 to p_3 instead of to p_2, which however allows a lower priority p_3 to proceed before a higher priority process p_2, resulting in priority inversion), the DAU may take an option to ask a lower priority process to release a resource involved in the situation so that the priority inversion may not occur (in Section 4.2.1, we provide Algorithm 6 for this case).

In this chapter, we use the same system model described by Section 1.4.1, Assumptions 1-8 of Section 1.5 and Assumptions 9 and 10 of Section 3.2.

4.2 Methodology

4.2.1 Our Deadlock Avoidance Method

Algorithm 3 shows our first new approach to deadlock avoidance. When a process requests a resource from the DAU (Line 2), the DAU checks for the availability of the resource requested (Line 3). If the resource is available (i.e., no process has currently been granted the resource), the resource will be granted to the requester immediately (Line 4; in Section 4.2.2 we will prove that no deadlock exists in this case). If the resource is not available, the DAU checks the possibility of request deadlock (R-dl in Definition 6) (Line 5). If the request would cause R-dl, the DAU does not accept the request (i.e., the request is denied); thus R-dl can be avoided (Line 6). On the other hand, if the request does not cause R-dl (Line 7), the DAU makes the request be pending since the resource is not available (Line 8).

When the DAU receives a resource release command from a process (Line 11), if no process is waiting for the resource (Line 18), the resource simply becomes available (Line 19). On the other hand, if a process is waiting for the resource, the DAU checks for the possibility of grant deadlock (G-dl in Definition 7) (Line 13) and next grants the resource to the requester only if the grant does not result in G-dl (Line 16). If, however, the grant would cause G-dl, the resource is not granted (Line 14).

Algorithm 3 Deadlock Avoidance Algorithm (DAA)

```
DAA (event) {
1   case (event) {
2       a request:
3           if the resource is available
4               grant the resource to the requester
5           else if the request would cause request deadlock (R-dl)
6               deny the request
7           else
8               make the request be pending
9           end-if
10          break;

11      a release:
12          if any process is waiting for the released resource
13              if the grant of the resource would cause grant deadlock
14                  do not grant the resource
15              else
16                  grant the resource to the process waiting
17              end-if
18          else
19              make the resource become available
20          end-if
21  } end-case
}
```

The above scheme is feasible and will avoid deadlock. Please note that this kind of scheme is similar to the Belik's approach [7] (see Section 2.2.4); the main difference is different methods and associated data structures for checking for cycles. Specifically, Belik's approach considers the deadlock avoidance problem as the problem of changing a directed acyclic graph while keeping it acyclic, whereas our approach focuses on iteratively discovering and removing removable edges to detect potential deadlock. Another difference is that Belik's approach requires compression (i.e., shrinking an $(m + n)$ by $(m + n)$ matrix to an m by n matrix via exploitation of the bipartite property of a resource allocation graph) to construct a "path matrix" (possibly enabling more efficient detection of a cycle), while our approach does not need any type of compression.

Algorithm 3, however, involves two drawbacks. One drawback is that, in Line 6 of

Algorithm 3, when a request is denied because of potential request deadlock (R-dl), the situation may introduce starvation of the processes involved in the potential R-dl (i.e., even though a system does not have a deadlock, no progress can be made by some processes, which is also known as livelock). In Section 4.2.2, we will prove that avoiding R-dl in this way can lead to livelock. The other drawback is that, in Line 14 of Algorithm 3, when a resource becomes available, if it cannot be granted because of grant deadlock (G-dl), not granting the resource can result in resource underutilization and/or livelock. Thus, the above scheme shown in Algorithm 3 is a good start yet requires some modification.

We propose three novel approaches to modify Algorithm 3 appropriately. Algorithm 4 implements an approach that avoids not only deadlock but also livelock associated with deadlock avoidance. When a request would cause R-dl (Line 5 of Algorithm 4), the request is denied with an error code telling the requester that it is potentially in R-dl (which may result in livelock as stated in the previous paragraph as the first drawback due to Line 6 of Algorithm 4) by setting the R-dl bit in a status register the requester reads. In this way, the requester is informed of potential livelock associated with deadlock avoidance; we assume that the requester voluntarily releases some resource(s) the requester holds in order to remove the possibility of livelock associated with deadlock avoidance.

In addition, when Algorithm 4 receives a resource release command from a process (Line 11 of Algorithm 4) and any process is waiting for the resource (Line 12), before actually granting the released resource to one of the requesters, Algorithm 4 temporarily marks a grant of the resource to the highest priority process (on its internal matrix). Then, to check potential grant deadlock, Algorithm 4 executes a deadlock detection algorithm. If the temporary grant does not cause grant deadlock (G-dl) (the "else" condition in Line 15), it becomes a fixed grant; thus the resource is granted to the highest priority requester (Line 16). On the other hand, if the temporary grant causes G-dl (Line 13), the temporary grant will be undone; then, because the released resource cannot be granted to the highest priority requester due to G-dl, Algorithm 4 tries to grant the resource to a lower

priority requester (Line 14). Algorithm 4 continues checking all processes to see if the released resource can be granted to a process without the involvement of deadlock (we will prove in Theorem 8 of Section 4.2.2 that there exists at least one process to which the released resource can be granted without G-dl). As a result, resources can be effectively exploited. Other behaviors are the same as Algorithm 3.

Algorithm 4 DAA (Approach Two)

```
DAA (event) {
1   case (event) {
2       a request:
3           if the resource is available
4               grant the resource to the requester
5           else if the request would cause request deadlock (R-dl)
6               deny the request and indicate R-dl is possible
                (this denial may result in possible livelock)
                (let the requester take care of this situation)
7           else
8               make the request be pending
9           end-if
10          break;

11      a release:
12          if any process is waiting for the released resource
13              if the grant of the resource would cause grant deadlock
14                  grant the resource to a lower priority process waiting
15              else
16                  grant the resource to the highest priority process waiting
17              end-if
18          else
19              make the resource become available
20          end-if
21  } end-case
}
```

While Algorithm 4 is a good strategy, it is somewhat passive since the resolution of livelock solely depends on the last requester having caused the potential request deadlock (R-dl), i.e., not considering the priorities of processes involved in the potential R-dl, the last requester needs to repeatedly rerequest and then finally may give up after a certain number of trials. For instance, if Algorithm 4 is employed in Example 1, which is an R-dl case, not

considering the priorities of VP and SP, SP needs to take appropriate action to resolve the potential R-dl since Algorithm 4 will inform SP of the potential R-dl because SP is the last requester. Additionally, the request case of Algorithm 4 does not consider the importance (i.e., priorities) of processes competing for resources. Thus, in order to more actively and efficiently resolve livelock, we propose another approach: Algorithm 5.

Algorithm 5 DAA (Approach Three)

```
DAA (event) {
1   case (event) {
2       a request:
3           if the resource is available
4               grant the resource to the requester
5           else if the request would cause request deadlock (R-dl)
6               if the priority of the requester greater than that of the owner
7                   make the request be pending
8                   ask the current owner of the resource to release the resource
9               else
10                  ask the requester to give up resource(s)
11              end-if
12          else
13              make the request be pending
14          end-if
15          break

16      a release:
17          the same as Algorithm 4
18  } end-case
}
```

As shown in Algorithm 5, if a request would cause request deadlock (R-dl) (Line 5 of Algorithm 5) – note that the DAU tracks all requests and releases – Algorithm 5 compares the priority of the requester with that of the current owner of the requested resource. If the priority of the requester is higher than that of the current owner of the resource (Line 6), Algorithm 5 makes the request be pending for the requester (Line 7), and then Algorithm 5 asks the owner of the resource to give up the resource so that the higher priority process can proceed (Line 8, the current owner may need time to finish or checkpoint its current processing). On the other hand, if the priority of the requester is lower than that of the

owner of the resource (Line 9), Algorithm 5 asks the requester to give up the resource(s) that the requester already has but is most likely not using yet (since all needed resources are not yet granted, Line 10). Other behaviors are the same as Algorithm 4.

Another drawback of Algorithm 4 is a potential priority inversion problem. Here what we mean by priority inversion may not be the common priority inversion problem due to critical section competition [44]. Instead, priority inversion in our context occurs due to resource competition where usage time of a resource may not be deterministic, as shown in the following example.

Example 23 If Algorithm 4 is employed in Example 4, at time t_5, instead of granting q_2 to p_2, by granting q_2 to p_3, the possible G-dl can be avoided by Algorithm 4. In this case, however, while p_3 proceeds, p_2 has to wait for q_2 until p_3 finishes using q_2. Since, in effect, p_3 is given priority over p_2 in this case, we denote this situation as priority inversion. ■

In some systems, such priority inversion can be as serious as deadlock. For such a system, we propose another algorithm, Algorithm 6. In Algorithm 6, in order to avoid priority inversion, whenever a resource is released and if any process is waiting (Line 5 of Algorithm 6), then the algorithm first grants the released resource to the highest priority process waiting (Line 6). Next, the algorithm checks grant deadlock (Line 7). If the grant has caused G-dl (Line 8), the algorithm asks the owner of a resource (say q_j) involved in the G-dl to release resource q_j so that once the owner releases resource q_j, then q_j will be granted to the highest priority process, which will be able to proceed, thereby resolving the G-dl as well as avoiding priority inversion.

Either Algorithm 4, Algorithm 5 or Algorithm 6 can potentially be employed in a system. For instance, Algorithm 4 can be used in a system that does not satisfy Assumption 6. On the other hand, Algorithm 6 is appropriate for a system that requires no priority inversion whatsoever. Nonetheless, we chose to implement Algorithm 5 in hardware because it resolves livelock more actively and efficiently than Algorithm 4 (in which the resolution of livelock depends on the last requester without considering priorities of processes

as mentioned in the next paragraph of Algorithm 4) and since Algorithm 5 resolves grant deadlock more quickly than Algorithm 6. As another usage, adoption of one or two flag(s) as parameter(s) of a user's choice (e.g., one parameter for a request choice and the other for a release choice) may enable users to select a most suitable algorithm (i.e., Algorithm 4, 5 or 6) that best fits to their specific target system.

Algorithm 6 DAA (Approach Four)

```
DAA (event) {
1   case (event) {
2       a request:
3           the same as Algorithm 5

4       a release:
5           if any process is waiting for the released resource
6               grant the resource to the highest priority process waiting
7               if the grant causes grant deadlock (G-dl)
8                   ask the current owner of a resource involved in the G-dl
                        to release the resource involved in the G-dl
9               end-if
10          else
11              make the resource become available
12          end-if
13  } end-case
}
```

Please note that our algorithms do not resolve all kinds of livelock defined in Definition 2. In fact, Algorithm 6 deals only with the case of livelock associated with deadlock avoidance. For example, starvation of a lower priority process due to the frequent executions of higher priority processes are not addressed by our approach.

4.2.2 Proof of the Correctness of the DAU

Proposition 1 *Given system γ_i, grant deadlock (G-dl in Definition 7) occurs when a request edge ($p_i \rightarrow q_j$) in state γ_{ij} is changed to a grant edge ($q_j \rightharpoonup p_i$) in state γ_{ik}, $j < k$, thereby forming a cycle in γ_{ik}.*

Proposition 2 *Given system γ_i, request deadlock (R-dl in Definition 6) occurs when a new*

request edge ($p_i \rightarrow q_j$) not in state γ_{ij} forms a cycle in state $\gamma_{i,j+1}$, where the request edge results in altering γ_{ij} into $\gamma_{i,j+1}$.

Lemma 7 *When a resource is available, an event of a request for the resource causes neither request deadlock nor grant deadlock if the resource is granted to the requester.*

Proof: If a resource q_j is currently available (unallocated), then by Assumption 8, q_j has currently neither any incoming edge nor any outgoing edge. Thus, a request from process p_i creates an edge $p_i \rightarrow q_j$. Since q_j has currently only one incoming edge, q_j cannot form part of a cycle; thus, q_j cannot be part of a deadlock.

After that, the request edge $p_i \rightarrow q_j$ is typically immediately changed to $q_j \rightharpoonup p_i$ since q_j is currently available. Now since q_j has only one outgoing edge, q_j cannot form part of a cycle; thus, q_j cannot be part of a deadlock. ■

Corollary 3 *Given system γ_i in system state γ_{ij}, when a resource node has two incoming edges and one outgoing edge, then there must exist at least two distinct paths in state γ_{ij}.*

Proof: Let the resource be q_1. Also let the two incoming edges be $e_1 = (p_1, q_1)$ and $e_2 = (p_2, q_1)$, and let the one outgoing edge be $e_3 = (q_1, p_3)$. Since there exist two distinct incoming edges, all three edges cannot be in one path because a node can appear at most once in a particular path (i.e., all nodes in a path must be distinct) by Definition 24. Thus, there exist at least two distinct paths in state γ_{ij} using edges e_1, e_2 and e_3. One such distinct path is a path with (p_1, q_1, p_3) and the other is a path with (p_2, q_1, p_3). ■

Corollary 4 *Given system γ_i in system state γ_{ij}, when a resource node has two incoming edges and one outgoing edge, if all these three edges are involved in deadlock, then there must exist at least two cycles in state γ_{ij}.*

Proof: Let the resource be q_1. Also let the two incoming edges be $e_1 = (p_1, q_1)$ and $e_2 = (p_2, q_1)$, and let the one outgoing edge be $e_3 = (q_1, p_3)$. Then, by Corollary 3, there

exist at least two paths as shown in the proof of Corollary 3. Also, by Theorem 4, if a system γ_i is in deadlock, there exists at least one cycle in state γ_{ij}.

Let us assume that the two paths form only one cycle. Since there exist two paths, for all three edges to be involved in deadlock, the cycle must include either (i) path ($p_1, q_1, p_3, \cdots, p_2, q_1, p_3$) or (ii) path ($p_2, q_1, p_3, \cdots, p_1, q_1, p_3$). However, both case (i) and case (ii) contradict the definition of path (Definition 24), since both paths include q_1 twice, a contradiction. Since there are no possibilities other than case (i) and case (ii), it cannot be true that the paths form only one cycle.

Thus, in order for two distinct paths to be involved in deadlock, there must exist at least two cycles. In other words, in order for a resource node with two incoming edges and one outgoing edge to be involved in deadlock, there must exist at least two cycles. ■

Theorem 7 *At an event of a request that causes request deadlock, denying the request results in livelock unless a process involved in the specific request deadlock releases a resource involved in the deadlock, assuming that processes involved in the deadlock repeatedly request resources until they acquire the resources.*

Proof: Request deadlock (R-dl) occurs when a path $p_i \leftharpoonup q_i \leftarrow p_{i+1} \leftharpoonup q_{i+1} \leftarrow \ldots p_{i+n} \leftharpoonup q_{i+n}$ forms a cycle as p_i requests one of the resources in the path, i.e., q_k where $i+1 \leq k \leq i+n$. Thus, denying the request $p_i \rightarrow q_k$ to avoid deadlock will result in p_i requesting q_k again after a certain amount of time; in fact, p_i repeatedly requests q_k until p_i acquires q_k under the assumption that no process in the path including p_i gives up or releases a resource in the cycle. By Definitions 1 and 2, p_i is in livelock and all other processes in the path are in deadlock. ■

Theorem 8 *For a system γ_i in state γ_{ij} not currently deadlocked, when a grant of a resource may occur either due to a release of the resource for which one or more process are waiting or due to a new request for the resource currently available, there must exist at least one process to which the resource can be granted without deadlock.*

Proof: A grant of a resource (say q_l) occurs in two cases: (i) resource q_l is currently available (unallocated) and the grant results from a new request; (ii) resource q_l has been allocated but just released, and the grant results from a pending request. Note here that no cycle (i.e., deadlock) has already formed; γ_{ij} is not currently deadlocked.

Case (i). By Lemma 7, the grant does not cause deadlock.

Case (ii). There are three sub-cases: (ii-1) one process is waiting for resource q_l, which has just been released; (ii-2) two processes are waiting for resource q_l; (ii-3) three or more processes are waiting for resource q_l.

In case (ii-1), since only one process (say p_i, i.e., $p_i \rightarrow q_l$) is waiting for q_l, which has just been released from p_i, edge $p_i \rightarrow q_l$ will be changed to $q_l \rightharpoonup p_i$; thus, q_l will have only one edge. Since q_l has only one edge, q_l cannot form part of a cycle; thus, q_l cannot be part of a deadlock.

In case (ii-2), let two processes be p_i and p_j, waiting for resource q_l, i.e., $p_i \rightarrow q_l \leftarrow p_j$. In this case, if the grant ($q_l \rightharpoonup p_i$) forms cycle(s), $p_j \rightarrow q_l$ must also be part of cycle(s) because q_l must be a connect node for there to be a cycle. Please note also that both p_i and p_j must also be connect nodes for there to be a cycle. Therefore, p_i must have at least one (possibly more) outgoing edge involved in the cycle(s), and p_j must have at least one (possibly more) incoming edge involved in the cycle(s) (implying that the grant $q_l \rightharpoonup p_i$ could cause many cycles). Here, rather than focusing on the fact that there can be many cycles, instead, let us pay more attention to the fact that p_j has one or more only incoming edges involved in cycle(s). In other words, the grant ($q_l \rightharpoonup p_i$) causing cycle(s) results from incoming edges of p_j (i.e., the direction of $\rightharpoonup p_j \rightarrow q_l \rightharpoonup p_i \rightarrow$). Hence, by granting q_l to p_j (i.e., $p_j \leftharpoondown q_l$) instead of p_i, cycle(s) will not form (i.e., $\rightharpoonup p_j \leftharpoondown q_l \leftarrow p_i \rightarrow$) because a cycle can form only one direction at a time and since there exists no cycle already formed.

There can be such a case that two processes and a resource are connected such a way that $p_i \rightarrow q_l \leftarrow p_j \rightarrow \cdots \rightharpoonup p_h \rightarrow \cdots \rightharpoonup p_i \rightarrow \cdots \rightharpoonup p_g \rightarrow \cdots \rightarrow p_j$. In this case, however, even though we exclude $\rightarrow q_l \leftarrow$, there has already a cycle formed such that

$p_j \rightarrow \cdots \rightharpoonup p_h \rightarrow \cdots \rightharpoonup p_i \rightarrow \cdots \rightharpoonup p_g \rightarrow \cdots \rightarrow p_j$ because here p_j appears twice in a path, which is the definition of a cycle (Definition 27). This fact contradicts our assumption that no cycle has already formed. Thus, this kind of special case is out of consideration.

Next, consider case (ii-3) where three processes (say p_i, p_j and p_k) are waiting for resource q_l. In this case, when q_l is given to p_i, q_l has two incoming edges and one outgoing edge. If the grant (i.e., $q_l \rightharpoonup p_i$) causes a cycle C_1, then either p_j or p_k or both must be involved in cycle C_1 because q_l must be a connect node for q_l to be involved in a cycle. Also, since q_l has two incoming edges, one or both of them must be in a cycle.

Let us consider the case where cycle C_1 is formed with p_j such that $p_i \leftharpoonup q_l \leftarrow p_j$. In this case, instead of granting q_l to p_i, but by granting q_l to p_j, cycle C_1 will not be formed as proven in case (ii-2). However, the grant $q_l \rightharpoonup p_j$ could also form a new cycle C_2 with p_k such that $p_j \leftharpoonup q_l \leftarrow p_k$. In this case, as is in the case of (ii-2), by granting q_l to p_k instead of p_j, both potential cycles C_1 and C_2 will not form and no cycles can form because a cycle can form only one direction at a time and since there exists no cycle already formed.

Let us now consider the grant $q_l \rightharpoonup p_i$ causes both p_j and p_k to be involved in cycles. Since there exist only one edge involved in cycle(s) between q_l and p_i (i.e., $q_l \rightharpoonup p_i$), all possible cycles must form in the direction of $q_l \rightharpoonup p_i$ unless any cycle has already formed. Therefore, either by granting q_l to p_j or p_k, there will not be any cycle because a cycle can form only one direction at a time and since no cycle has already formed.

In the same way, in cases where four or more processes are waiting for a released resource, there must be a process (waiting for the released resource) to which the resource can be given without deadlock because a cycle can form only one direction at a time and since there exists no cycle already formed. ■

4.2.3 Run-time Complexity of the DAU

The DAU becomes active and starts working only when a request or a release event occurs. Once the DAU is activated, it operates in at most $2 \times n \times min(m, n) + (n \times k)$ clock

cycles, where m and n are the numbers of resources and processes, respectively, and where k is the number of cycles that each trial of unsuccessful grants takes (except each deadlock check at each trial), stated in Line 14 of Algorithm 4. If it is assumed that $min(m, n) \gg k$, then the run-time complexity of the DAU becomes $O(n \times min(m, n))$. This run-time is a theoretical lower bound on the complexity and is valid as long as the clock period is longer enough than the maximum delay of a critical calculation. After finishing operation, the DAU remains idle until a next event occurs. Processes and the DAU communicate via specific application programming interfaces (APIs).

4.3 Implementation

4.3.1 Architecture of the DAU

Figure 30 illustrates the DAU architecture. The DAU consists of four parts: a DDU, command registers (one for each process), status registers (one for each process) and a unit implementing Algorithm 5 with a finite state machine. The DDU architecture was already described in Section 3.3.4. The DAA logic mainly controls the DAU behavior, i.e., interprets and executes commands (requests or releases) from PEs, and returns processing results back to PEs via status registers. Command registers receive commands from each PE. The command results of the DAU are stored into status registers read by all PEs.

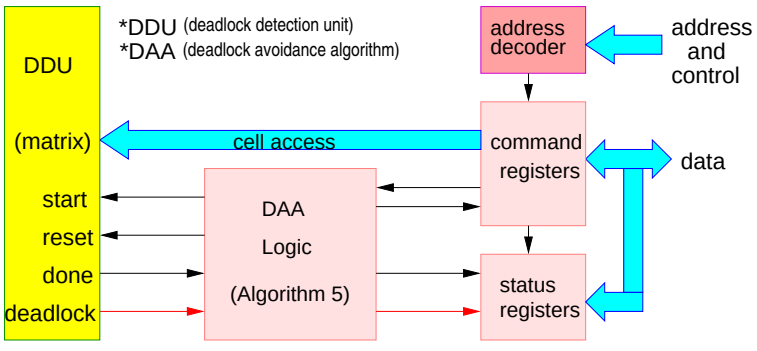


Figure 30: DAU architecture.

As shown in Figure 31, a command register contains fields *Release*, *Request* and *Resource*; *Resource* indicates resources being requested or released (one-hot encoded). If Bit1 is a '0', then Bit0 must be a '1', and the result is a command requesting the resource(s) specified by Bits16-31. If Bit1 is a '1', then Bit0 must be a '0', and the result is a command releasing the resource(s) specified by Bits16-31. Please note that Bits16-31 are one-hot encoded so that, in this specific case, any combination of 16 resources can be released or requested with a single command.

A status register contains necessary information such as *done, busy, successful, pending, give-up, invalid* as well as *G-dl* and *R-dl* as shown in Figure 32. *Done* indicates that the DAU has just finished processing an event and the status result is valid. *Busy* indicates that the DAU is processing an event. *Successful* signifies that the command has successfully processed. *Pending* notifies a process reading this status register that the process has pending request(s). *Give-up* asks a process to give up a resource specified by Bits12-15 of Figure 32. *Invalid* means that the command is invalid or unknown. *G-dl* represents a potential grant deadlock. *R-dl* informs a requester of potential request deadlock. A status register shown in Figure 32 also contains *Assigned* (Bits16-31) and *Resource* (Bits12-15). *Assigned* indicates which resources have been allocated using a one-hot encoding. *Resource* in Figure 32 signifies a binary coded value of a single resource (note that 4 bits interpreted as an unsigned binary number yield 16 possibilities) that a process is requested to give up.

Bits	Name	Description
Bit0:	Request	The process is requesting a resource specified by Bit16-31.
Bit1:	Release	The process is releasing a resource specified by Bit16-31.
Bit16-31:	Resource	A resource being requested or released (one-hot encoded).

Figure 31: DAU command register.

0	1	2	3	4	· · ·	31

Bits	Name	Description
Bit0:	G-dl	Potential grant deadlock is detected.
Bit1:	R-dl	Potential request deadlock is detected.
Bit2:	Busy	Unit is processing an event.
Bit3:	Done	Unit has just finished the process of an event and the status result is valid.
Bit4:	Grant	The request is granted.
Bit5:	Successful	The command has successfully processed.
Bit6:	Invalid	The command is invalid or unknown.
Bit7:	Pending	The process reading this status register has pending request(s).
Bit8:	Give_up	The process needs to give up a resource specified by Bit12-15.
Bit12-15:	Resource	A binary coded value of a resource that a process needs to give up.
Bit16-31:	Assigned	Resources already assigned and thus unavailable (one-hot encoded).

Figure 32: DAU status register.

Furthermore, in Bits0-8 of a DAU status register (Figure 32), a value of a bit = '1' indicates a command or a status is active.

Note that Bits2-15 of a DAU command register and Bits9-11 of a DAU status register are reserved for future use; e.g., while currently at most 16 resources can be specified, these bits reserved for future use could be used, for example, to allow more resources to be specified. Finally, please note that more than 32 bits could be used for the DAU command and DAU status registers with a corresponding cost in terms of I/O hardware and software design.

4.3.2 Synthesized Result of the DAU

We use the Synopsys Design Compiler [52] to synthesize various DAU sizes with the Qual-Core Logic .25μm standard cell library [39]. The Synthesis result is shown in Table 4. The "Total Area" column denotes the area in units equivalent to a minimum-sized two-input NAND gate in the library. DAU5x5 represents a DAU for five processes and five resources.

In case where an MPSoC contains four PowerPC 755 PEs (1.7M gates each) and 16MB memory (33.5M gates), the resulting MPSoC area, the sum of area of 16MB memory plus four MPC755's plus DAU20x20 (i.e., 33.5M + 1.7M $\times$ 4 + 15247), is 40315247 gates. Thus, the area overhead in the MPSoC due to the DAU 20x20, i.e., the area of DAU20x20 divided by the total MPSoC area is approximately .04% (i.e., $15247/40315247$).

Table 4: Synthesized result of the DAU.

Module Name	Lines of Verilog	Total Area
DAU5x5	523	1597
7x7	552	2429
10x10	612	4309
15x15	753	8868
20x20	943	15247
MPSoC w/ DAU20x20	–	40.32M

4.4 Experiments

4.4.1 Simulation Environment Setup for the DAU evaluation

The experimental simulations evaluating the DAU performance were carried out using Seamless Co-Verification Environment (CVE) [33] aided by Synopsys VCS [53] for Verilog HDL simulation and XRAY [34] for software debugging. We use Atalanta RTOS version 0.3 [51], a shared-memory multiprocessor RTOS. The other simulation setups not mentioned here such as a bus clock rate and a system memory size are the same in Section 3.4.1.

For the DAU experimental simulations, we use the same MPSoC introduced in Section 3.4.1 and in Figure 27 but with the DAU instead of the DDU; please see Figure 33. The MPSoC has a DAU for five processes and five resources. We invoke one process on each PE and prioritize all processes, p_1 being the highest and p_4 being the lowest.

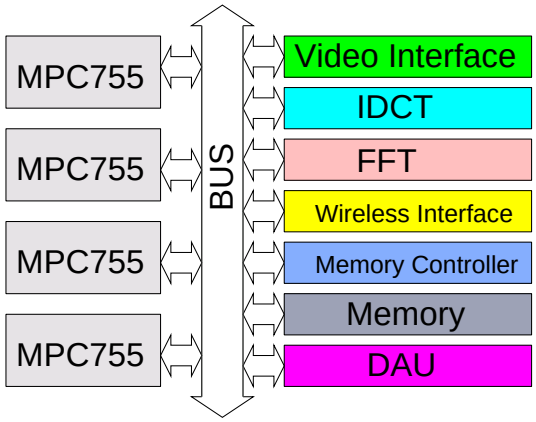


Figure 33: MPSoC architecture for the DAU evaluation.

4.4.2 Application Example I

For this experiment we utilize the DAU implementing Algorithm 5. We show a sequence of requests and grants that would lead to grant deadlock (G-dl) as shown in Figure 34 and Table 5. Recall that there is no constraint on the ordering of resource usage. That is, when a process requests a resource and the resource is available, it is granted immediately to the requesting process. At time t_1, process p_1, running on PE1, requests both Video Interface (VI) and Inverse Discrete Cosine Transform (IDCT), which are then granted to p_1. After that, p_1 starts receiving a video stream through VI and performs IDCT processing. At time t_2, process p_3, running on PE3, requests IDCT and Wireless Interface (WI) to convert a frame to an image and to send the image through WI. However, only WI is granted to p_3 since IDCT is unavailable. At time t_3, p_2 running on PE2 also requests IDCT and WI, which are not available for p_2. When IDCT is released by p_1 at time t_4, IDCT would typically (assuming the DAU is not used) be granted to p_2 since p_2 has a priority higher than p_3; thus, the system would typically end up in deadlock. However, the DAU checks the potential G-dl and then avoids the G-dl by granting IDCT to p_3 even though p_3 has a priority lower than p_2. Then, p_3 uses and releases IDCT and WI at time t_6. After that, IDCT and WI are granted to p_2 at time t_7, which finishes its job at time t_8.

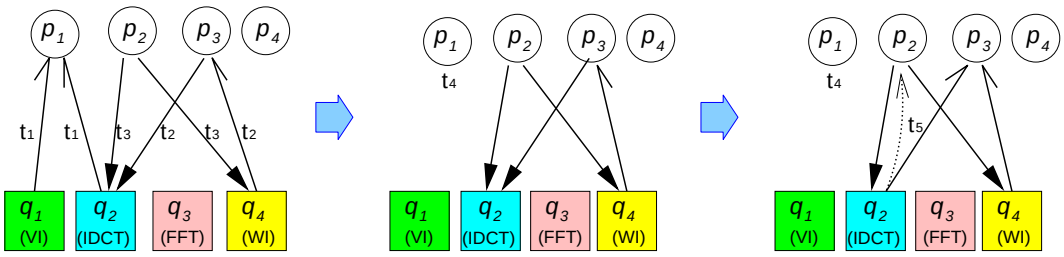


Figure 34: Events RAG for grant deadlock avoidance comparison.

Table 5: A sequence of requests and grants that would lead to G-dl.

Time	Events
t_0	The application starts.
t_1	p_1 requests q_1 and q_2, which are granted to p_1 immediately.
t_2	p_3 requests q_2 and q_4; only q_4 is granted to p_3 since q_2 is not available.
t_3	p_2 also requests q_2 and q_4.
t_4	q_1 and q_2 are released by p_1.
t_5	Then, the DAU tries to grant q_2 to p_2 since p_2 has a priority higher than p_3. However, the DAU detects potential G-dl. Thus, the DAU grants q_2 to p_3, which does not lead to a deadlock.
t_6	q_2 and q_4 are used and released by p_3.
t_7	q_2 and q_4 are granted to p_2.
t_8	p_2 fi nishes its job, and the application ends.

With the above scenario, we wanted to measure two figures, the average execution time of the deadlock avoidance algorithm used and the total execution time of the application in two cases: (i) using the DAU versus (ii) using DAA (Algorithm 5) in software.

4.4.3 Experimental Result for Application Example I

Table 6 shows that the DAU achieves a 312X speedup of the average algorithm execution time and gives a 37% speedup of application execution time over avoiding deadlock with DAA in software. Algorithm 5 in software. Please note that during the run-time, the application invoked deadlock avoidance 12 times (since every request and every release invokes the deadlock avoidance algorithm in use).

Table 6: Execution time comparison (G-dl case 1).

Method of Implementation	Algorithm Run Time*	Application Run Time*	Speedup[&]
DAU(hardware)	7	34791	$\frac{47704-34791}{34791} = 37\%$
DAA in software	2188	47704	

*The time unit is a bus clock (10 ns), and the values are averaged.
[&]The speedup is calculated according to the formula by Hennessy and Patterson [18].

4.4.4 Application Example II and Its Result

For the second experimentation, we utilize the DAU implementing Algorithm 6 with the same scenario as Application Example I (Section 4.4.2). In case of the grant deadlock which appears at time t_5 in Table 5, after granting q_2 to p_2 (since p_2 has a priority higher than p_3), Algorithm 6 avoids deadlock by asking p_3 to give up resource q_4, thereby finishing the application without deadlock. Specifically, when q_2 is released by p_1, the algorithm first grants q_2 to p_2 since p_2 has a priority higher than p_3 although both p_2 and p_3 are waiting for q_2. Then, the algorithm detects grant deadlock and thus asks p_3 to release q_4 by sending a *Give-up* signal (described in Section 4.3.1). Once p_3 voluntarily releases q_4, the algorithm receives the release of q_4 and grants q_4 to p_2 since p_2 is the highest priority process waiting for q_4. Therefore, p_2 proceeds and p_3 will proceed after p_2.

Table 7 shows that the DAU provides a 305X speedup of the average algorithm execution time and achieves a 38.5% speedup of application execution time over avoiding deadlock with Algorithm 6 in software. Please note that the time between when p_3 is asked to release q_4 and when p_3 actually releases q_4 is 156 clock cycles using the DAU and 724 clock cycles using Algorithm 6 in software. Please note also that the time between when p_3 is asked to release q_4 and when q_4 is granted to p_2 took 159 clock cycles using the DAU, while the same period took 1556 clock cycles using Algorithm 6 in software.

Table 7: Execution time comparison (G-dl case 2).

Method of Implementation	Algorithm Run Time*	Application Run Time*	Speedup
DAU(hardware)	7.1	34890	$\frac{48324-34890}{34890} = 38.5\%$
DAA in software	2165	48324	

*The time unit is a bus clock, and the values are averaged.

4.4.5 Application Example III

For this experiment we utilize the DAU implementing Algorithm 5. We show a sequence of requests and grants that would lead to request deadlock (R-dl) as shown in Figure 35. In this example, we assume the following. (i) Process p_1 requires resources q_1 (VI) and q_2 (IDCT) to complete its job. (ii) Process p_2 requires resources q_2 (IDCT) and q_3 (FFT). (iii) Process p_3 requires resources q_3 (FFT) and q_1 (VI). The detailed sequence is shown in Table 8.

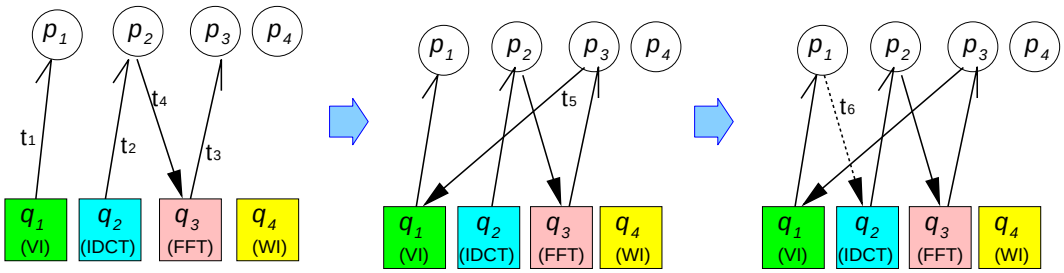


Figure 35: Events RAG for request deadlock avoidance comparison.

Let us now explain the sequence of events in Table 8. At time t_1, process p_1 requests and acquires q_1. At time t_2, process p_2 requests and acquires q_2. At time t_3, process p_3 requests and acquires q_3. After that, at time t_4, process p_2 requests q_3; since q_3 was already granted to p_3, and since the request does not cause R-dl, the request becomes pending. At time t_5, process p_3 requests q_1; since q_1 was already granted to p_1, and since the request does

not cause R-dl, this request also becomes pending. At time t_6, when process p_1 requests q_2, request deadlock (R-dl) would occur. However, the DAU detects the potential R-dl and then avoids the R-dl by asking p_2 to give up resource q_2 since p_1 has a priority higher than p_2, which is the current owner of q_2. As a result, at time t_7, p_2 gives up and releases q_2, which is going to be granted to p_1 (of course, p_2 has to request q_2 again). After using q_1 and q_2, p_1 releases q_1 and q_2 at time t_8. While q_1 is going to be granted to p_3, q_2 is going to be granted to p_2. Thus, p_3 uses q_1 and q_3 and then releases q_1 and q_3 at time t_9; q_3 is granted to p_2, which then uses q_2 and q_3 and finishes its job at time t_{10}.

Table 8: A sequence of requests and grants that would lead to R-dl.

Time	Events
t_0	The application starts.
t_1	p_1 requests q_1; q_1 is granted to p_1.
t_2	p_2 requests q_2; q_2 is granted to p_2.
t_3	p_3 requests q_3; q_3 is granted to p_3.
t_4	p_2 requests q_3, which becomes pending.
t_5	p_3 requests q_1, which also becomes pending.
t_6	p_1 requests q_2, which is about to lead to R-dl. However, the DAU detects the possibility of R-dl. Thus, the DAU asks p_2 to give up resource q_2.
t_7	p_2 releases q_2, which is granted to p_1. A moment later, p_2 requests q_2 again.
t_8	p_1 uses and releases q_1 and q_2. Then, while q_1 is granted to p_3, q_2 is granted to p_2.
t_9	p_3 uses and releases q_1 and q_3, which are granted to p_2.
t_{10}	p_2 finishes its job, and the application ends.

We similarly measured two figures, the average execution time of deadlock avoidance algorithms and the total execution time of the application in two cases: (i) exploiting the DAU and (ii) using Algorithm 5 in software.

4.4.6 Experimental Result for Application Example III

Table 9 demonstrates that the DAU achieves a 294X speedup of the average algorithm execution time and gives a 44% speedup of application execution time over avoiding deadlock with Algorithm 5 in software. Please note that during the run-time, the application invoked deadlock avoidance 14 times.

Table 9: Execution time comparison (R-dl).

Method of Implementation	Algorithm Run Time*	Application Run Time*	Speedup
DAU(hardware)	7.14	38508	$\frac{55627-38508}{38508} = 44\%$
DAA in software	2102	55627	

*The time unit is a bus clock, and the values are averaged.

Please note also that in systems where events of resource requests and releases occurs relatively rarely compared to computation and processing time of an actual application, such dramatic performance improvement shown in our experiment may not be achieved.

4.5 Summary

Several variants of a novel Deadlock Avoidance Algorithm (DAA) and a hardware implementation in the Deadlock Avoidance Unit (DAU) are described in this chapter. The DAU provides a very fast and very low area way of avoiding deadlock at run-time, which helps free programmers from worrying about deadlock.

We demonstrate the following through experimentation: (i) The DAU automatically avoids deadlocks as well as reduces the deadlock avoidance time by 99% *(about 300X)* as compared to DAA in software. (ii) The DAU achieves in a particular example a 44% speedup of application execution time as compared to the execution time of the same application that uses DAA in software. While our examples are not industrial strength full product code, nevertheless we expect similar results as MPSoC designs become more commonplace; we predict that our DAU can potentially help especially in real-time scenarios where at time-critical moments significant transitions involving many releases, requests and grants occur.

In the next chapter, a parallelized version of the Habermann's Banker's Algorithm and its hardware implementation (i.e., the Parallel Banker's Algorithm Unit) will be explained.

CHAPTER V

PARALLEL BANKER'S ALGORITHM UNIT

5.1 Introduction

Given the current System-on-a-Chip (SoC) technology trends discussed in Section 2.1, we predict that in the near future, MultiProcessor System-on-a-Chip (MPSoC) designs will, as shown in Figure 36, have many Processing Elements (PEs) and hardware resources including various multiple-instance resources.

A multiple-instance resource typically has multiple hardware blocks of the same or similar functionality available to all processors for the purpose of easy management or due to the natural structure of a resource. Examples of such multiple-instance resources include a multiple reenterable lock, a counting semaphore (but not binary semaphores) [31], a group of blocks of memory or input/output buffers, and a group of communication channels, to just name a few. In addition, one resource would be considered as a multiple-instance resource if it could process multiple blocks of data for multiple processes at the same time. Possible examples of this kind of multiple-instance resource are pipelined DSP processors [30] and pipelined MPEG encoder/decoders [6]. Furthermore, to increase system performance when a system has two or more hardware resources that provide the same functionality, then these resources can also be considered as a multiple-instance resource such as a TMS320C80 chip (consisting of four DSP processors in a single silicon die from Texas Instruments [54]). We believe that there will be such resources in an MPSoC in the future, resulting in resource allocation problems.

We envision that one important way of supporting high levels of concurrency is to handle deadlock problems in such systems so that programmers and users do not have to worry about the freezing of their systems because of deadlock. Thus, we propose a novel

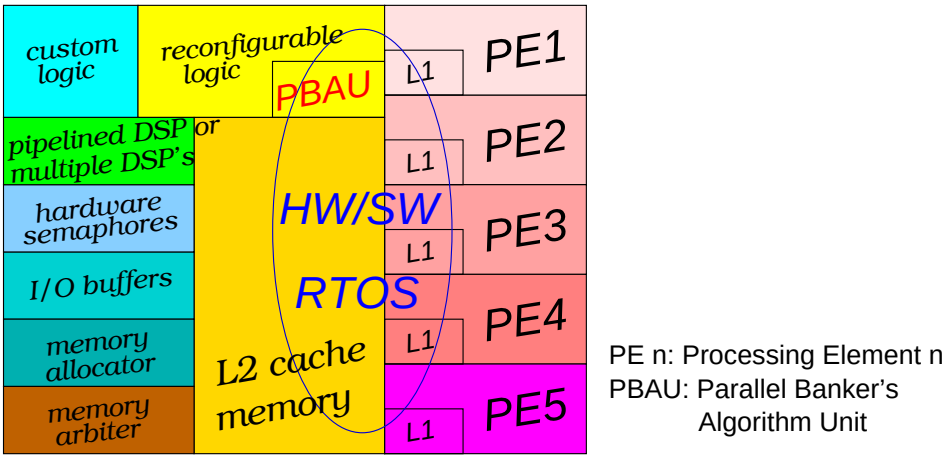


Figure 36: MPSoC with multiple-instance resources.

Parallel Banker's Algorithm (PBA), implement PBA in Verilog HDL and demonstrate its performance evaluation so that MPSoC programmers, who are reluctant to exploit deadlock avoidance approaches even as such approaches increase in importance, may be willing to adopt a faster hardware version of a deadlock avoidance approach.

The fundamental deadlock avoidance approach is the well-known Banker's Algorithm (BA) in the operating system realm. Dijkstra first introduced BA for single multiple-instance resource systems [11], and later Habermann improved BA to be able to handle multiple-instance multiple-resource systems [16]. In BA, each process declares the maximum possible number of instances for each resource it may need. Given this information, as each resource request is made, an assignment is authorized provided that there exists at least one sequence of executions that does not evolve to deadlock. The run-time complexity of Habermann's BA in software is $O(m * n^2)$, where m and n are the numbers of resources and processes, respectively. The efficiency of the algorithm was later improved to $O(m * n)$ by Holt [20]. However, Holt's algorithm is costly and disadvantageous to implement in hardware in terms of maintaining ordered lists of all requests.

Even though BA was proposed a few decades ago, minor variations to BA are still being proposed for critical systems that can greatly benefit from the algorithm. For instance,

in 2001, Gebraeel and Lawley applied BA to automated tool sharing systems [14], and, in 2002, Ezpeleta et al. proposed a banker's solution for deadlock avoidance in flexible manufacturing systems [13].

Because the DAU presented in Chapter 4 is implemented based on a Resource Allocation Graph (RAG) approach for single-instance resources, the DAU can only be used for systems exclusively with single-instance resources. The Parallel Banker's Algorithm Unit (PBAU) [25], on the contrary, can be used for not only a system with single-instance resources but also a system with multiple-instance resources as well.

5.2 Target System Model

To describe our system model for PBAU, we show in the following example a possible MPSoC target, which is a slightly modified version of the target MPSoC shown in Section 1.4.1.

Example 24 A future MPSoC with multiple-instance resources

We refer to the device shown in Figure 36 as a particular MPSoC example. This MPSoC consists of five Processing Elements (PEs) and three resources – a counting semaphore with a group of I/O buffers, another counting semaphore with a group of multiple DSP processors, and an SoCDMMU memory allocator [46] with a large L2 memory. Counting semaphores [11] are used to manage limited resources (including managing access to the resources). The MPSoC also contains a memory arbiter and a PBAU. PBAU in Figure 36 receives all requests and releases, decides whether or not the request can cause a deadlock and then permits the request only if no deadlock results (i.e., the system remains safe). ■

We consider this kind of request-grant system with many resources and PEs shown in Figure 36 as our system model for PBAU.

5.3 Methodology

5.3.1 Usage Assumption

If PBAU is employed, all processes have to request or release resources through PBAU; thereby PBAU tracks all requests and releases of resources. In other words, PBAU receives and interprets a command from a process; then, after necessary processing, such as executing PBA when the command is a request, the PBAU returns a command result back to the process to indicate whether the release or request is successful and/or acceptable.

5.3.2 Our Deadlock Avoidance Method

This section explains the main concept of our novel Parallel Banker's Algorithm (PBA [25]). Algorithm 7 shows PBA for multiple-instance multiple-resource systems. PBA executes whenever a process is requesting resources and returns the status of whether the request is successfully granted or rejected due to the possibility of deadlock. PBA decides if the system is still going to be sufficiently safe after the grant, i.e., if there exists at least a sequence of process executions without deadlock after some allocation of the resources that a process requested.

Before explaining the details of PBA, let us first introduce notations used in this chapter as shown in Table 10 and data structures as shown in Table 11. Request[i][j] is a request for resource j from process i. If resource j is a single-instance resource, Request[i][j] is either '0' or '1'; otherwise, if resource j is a multiple-instance resource, Request[i][j] can take on values greater than one. Maximum[i][j] represents the maximum instance demand of process i for resource j. Available[j] indicates the number of available instances of resource j. Allocation[i][j] records the number of instances of resource j allocated to process i. Need[i][j] contains the number of additional instances of resource j that process i may need. Work[] (i.e., Work[j] for all j) is a temporary storage for Available[] (i.e., Available[j] for all j). Finish[i] denotes the potential completeness of process i. Wait_count[i] is a counter for each process that is incremented by one each time a request is denied; proper

use of Wait_count[i] can enable some known livelock situations to be broken.

All variables containing resource information in our experimentation are 4-bit values, which means that our implementation supports up to 16 instances for each resource. However, this can easily be extended. By parameterized generation of the PBAU described in Section 6.3, any PBAU size and any number of instances can be supported.

Table 10: Notations for PBA.

notation	explanation
p_i	a process
q_j	a resource
array[][] or array[]	all elements of the array
array[i][]	all elements of row i of the array
array[][j]	all elements of column j of the array

Table 11: Data structures for PBA.

name	notation	explanation
Request[i][j]	R_{ij}	request from process i for resource j
Maximum[i][j]	X_{ij}	maximum demand of process i for resource j
Available[j]	V_j	current number of unused resource j
Allocation[i][j]	G_{ij}	process i's current allocation of j
Need[i][j]	N_{ij}	process i's potential for more j (Need[i][j]=Maximum[i][j]-Allocation[i][j])
Work[j]	W_j	a temporary storage (array) for Available[j]
Finish[i]	F_i	potential completeness of process i
Wait_count[i]	C_i	wait count for process i to break livelock

PBA takes as input the maximum requirements of each process and guarantees that the system always remains in an H-safe state. Tables (data structures or arrays) are maintained of available resources, maximum requirements, current allocations of resources and resources needed, as shown in Table 11. PBA uses these tables/matrices to determine whether the state of the system is either H-safe or H-unsafe. When resources are requested by a process, the tables are updated *pretending* the resources were allocated. If the tables

will result in an H-safe state, then the request is actually granted; otherwise, the request is not granted, and the tables are returned to their previous states.

Algorithm 7 Parallel Banker's Algorithm (PBA)

PBA (Process i sends Request[i][] for resources) {
1 STEP 0: p_i sends Request[i][] for resources
2 STEP 1: if ($\forall j$, (Request[i][j] $\leq$ Need[i][j])) /* $\forall$ means *for all*. */
3 goto STEP 2
4 else ERROR
5 STEP 2: if ($\forall j$, (Request[i][j] $\leq$ Available[j]))
6 goto STEP 3
7 else deny p_i's request, increment Wait_count[i] by one and return
8 STEP 3: pretend to allocate requested resources
9 $\forall j$, Available[j] := Available[j] – Request[i][j]
10 $\forall j$, Allocation[i][j] := Allocation[i][j] + Request[i][j]
11 $\forall j$, Need[i][j] := Maximum[i][j] – Allocation[i][j]
12 STEP 4: prepare for the H-safety check
13 $\forall j$, Work[j] := Available[j]
14 $\forall i$, Finish[i] := false
Let *able-to-finish(i)* be ((Finish[i] == false) and ($\forall j$, Need[i][j] $\leq$ Work[j]))
15 STEP 5: Find all i such that *able-to-finish(i)*
16 if such i exists,
17 $\forall j$, Work[j] := Work[j] + $\mathbf{\Sigma}_{i \text{ such that } \textit{able-to-finish(i)}}$ Allocation[i][j]
18 $\forall i$, if *able-to-finish(i)* then Finish[i] := true
19 repeat STEP 5
20 else (i.e., no such i exists) goto STEP 6 (end of iteration)
21 STEP 6:
22 if ($\forall i$, (Finish[i] == true))
23 then pretended allocations anchor; p_i proceeds (i.e., H-safe)
24 else
25 restore the original state and deny p_i's request (i.e., H-unsafe)
}

Let us explain Algorithm 7 step by step. A process can request multiple resources at a time as well as multiple instances of each resource. In Step 1 (Line 2), when a process requests resources, PBA first checks if the request does not exceed Need[i][] for the process. If the request is within its pre-declared claims, in Step 2 (Line 5) PBA checks if there are sufficient available resources for this request. If sufficient resources exist, PBA continues to Step 3; otherwise, the request is denied and the value of the wait counter (in variable Wait_count[i] of Table 11) for the process increases by one to break a possible livelock

if necessary (e.g., if a request from a process is denied more than a threshold number of denial times, a process may release resources the process holds or take appropriate action assuming that there may exist a possible livelock – please see Section 5.3.5 for an extended discussion of this case).

In Step 3 (Lines 8-11), it is pretended that the request could be fulfilled, and the tables are temporarily modified according to the request.

In Step 4 (Lines 12-14), PBA prepares for the H-safety check, i.e., initializes variables Finish[] and Work[]. Work[] is used to search processes that can finish their jobs by using both resources currently Available[] and resources which will become available during the execution of an H-safe sequence (i.e, resources currently held by previous processes in a H-safe sequence, please see Definition 8).

In Step 5 (Lines 15-20), PBA finds processes that can finish their jobs by acquiring some or all resources available according to Work[] (please see the previous paragraph). If one or more such processes exist, PBA adds all resources that these processes hold to Work[], then declares these processes to be *able-to-finish* (i.e., Finish[i] := true for each process i), and finally repeats Step 5 until all processes can finish their jobs. On the other hand, if no such process exists – meaning either all processes became *able-to-finish* or no more processes can satisfy the comparison (i.e., Need[i][j] $\leq$ Work[j] for all j) – PBA moves to Step 6 to decide whether or not the pretended allocation state is H-safe.

In Step 6 (Lines 21-25), if all processes have been declared to be *able-to-finish*, then the pretended allocation state is in an H-safe state (meaning there exists an identifiable H-safe sequence by which all processes can finish their jobs in the order of processes having been declared to be *able-to-finish* in the iterations of Step 5); thus, the requester can safely proceed. However, in Step 6, if there remain any processes unable to finish, the pretended allocation state may cause deadlock; thus, PBA denies the request, restores the original allocation state before the pretended allocation and also increases the wait count for the requester.

The following example illustrates how PBA works in a simple yet general case.

Example 25 An example of resource allocation controlled by PBA

Consider a system with three processes p_1, p_2 and p_3 and two resources q_1 and q_2, where q_1 has three instances and q_2 has two instances. Table 12 shows a possible current resource allocation status in the system as well as maximum resource requirements for each process. Notice that Need[i][j] = Max[i][j] - Allocation[i][j].

Table 12: A resource allocation state.

	Maximum	Allocation	Need	Available
	q_1 q_2	q_1 q_2	q_1 q_2	q_1 q_2
p_1	3 2	1 1	2 1	1 1
p_2	2 1	1 0	1 1	
p_3	1 2	0 0	1 2	

Currently one instance of q_1 and one instance of q_2 are given to p_1, and another instance of q_1 is given to p_2. Thus, only one instance of q_1 and one instance of q_2 are available. At this moment, let us consider two cases. i) When p_2 requests one instance of q_1, will it be safely granted? ii) When p_1 requests one instance of q_2, will it be safely granted? In case i), let us pretend to grant q_1 to p_2; then the allocation table would be changed as shown in Table 13.

Table 13: Initial resource allocation state for case i).

	Maximum	Allocation	Need	Available
	q_1 q_2	q_1 q_2	q_1 q_2	q_1 q_2
p_1	3 2	1 1	2 1	0 1
p_2	2 1	2 0	0 1	
p_3	1 2	0 0	1 2	

Now PBA checks if the resulting system stays in an H-safe state (see Theorem 1). That is, there must exist an H-safe sequence even if all processes were to request their maximum needs after the pretended grant [11, 16, 20]. The following corresponds to Step 5 of PBA. From Table 13, if p_2 requests one more instance of q_2 (i.e., up to p_2's maximum claim), since q_2 is available, q_2 is going to be granted to p_2, which will finally finish its job and release all resources. Then, the available resources will be two instances of q_1 and one instance of q_2 as shown in Table 14.

Next, p_1 can acquire these available resources, finish its job and release all resources; the available resources will be three instances of q_1 and two instances of q_2 as shown in Table 15.

Table 14: Resource allocation state in case i) after p_2 finishes.

	Maximum	Allocation	Need	Available
	q_1 q_2	q_1 q_2	q_1 q_2	q_1 q_2
p_1	3 2	1 1	2 1	2 1
p_2	2 1	0 0	2 1	
p_3	1 2	0 0	1 2	

Similarly, p_3 can acquire these available resources and finally finish its job. As a result, an H-safe sequence exists in the order p_2, p_1 and p_3. That is, after the grant of q_1 to p_2, the system remains in an H-safe state.

Table 15: Resource allocation state in case i) after p_1 finishes.

	Maximum	Allocation	Need	Available
	q_1 q_2	q_1 q_2	q_1 q_2	q_1 q_2
p_1	3 2	0 0	3 2	3 2
p_2	2 1	0 0	2 1	
p_3	1 2	0 0	1 2	

Now considering case ii), let us pretend to grant q_2 to p_1; then the allocation table would be changed as shown in Table 16 (which is appropriately altered from Table 12). From this moment on, neither processes p_1, p_2 nor p_3 can acquire up to its declared maximum unless another process releases resources that the process holds. Thus, the system will not remain in an H-safe state. As a result, the algorithm will deny the request in case ii). ■

Table 16: A resource allocation state in case ii).

	Maximum	Allocation	Need	Available
	q_1 q_2	q_1 q_2	q_1 q_2	q_1 q_2
p_1	3 2	1 2	2 0	1 0
p_2	2 1	1 0	1 1	
p_3	1 2	0 0	1 2	

The gist of our approach is that because the operations in Step 5 are performed in parallel, if Need[i][j] $\leq$ Work[j] for all i and for all j are satisfied at the first iteration, PBA finishes at once, resulting in $O(1)$ run-time. Such an example is described in Example 26.

Example 26 An example of resource allocation in a special case

Consider the same system as Example 25 with three processes p_1, p_2 and p_3 and two resources q_1 and q_2, but in this example q_1 has five instances and q_2 has four instances. Table 17 shows a possible current resource allocation state in the system as well as maximum resource requirements for each process.

Table 17: A resource allocation state in a special case.

	Maximum	Allocation	Need	Available
	q_1 q_2	q_1 q_2	q_1 q_2	q_1 q_2
p_1	3 2	1 1	2 1	3 3
p_2	2 1	1 0	1 1	
p_3	1 2	0 0	1 2	

Currently one instance of q_1 and one instance of q_2 are given to p_1, and another instance of q_1 is given to p_2. Thus, three instances of q_1 and three instances of q_2 are available. At this moment, if p_1 requests one instance of q_1 and one instance of q_2, will q_1 and q_2 be safely granted to p_1? Since the request is within the need of p_1 and the availability of resources (corresponding to Steps 1 and 2 in Algorithm 7), PBA proceeds to pretend to grant one q_1 and one q_2 to p_1 (Step 3); then the allocation table would be changed as shown in Table 18.

Table 18: A resource allocation state after pretense.

	Maximum	Allocation	Need	Available
	q_1 q_2	q_1 q_2	q_1 q_2	q_1 q_2
p_1	3 2	2 2	1 0	2 2
p_2	2 1	1 0	1 1	
p_3	1 2	0 0	1 2	

Now PBA checks if the resulting system stays in an H-safe state. The following corresponds to Step 5 of PBA. From Table 18, if p_1 requests one more instance of q_1 (i.e., up to its maximum claim), since q_1 is available, q_1 can be given to p_1; thus, p_1 can finish its job. If p_2 requests one instance of q_1 and one instance of q_2 (i.e., up to its maximum claim), since q_1 and q_2 are available, both can be given to p_2; thus, p_2 can finish its job. In a similar fashion, p_3 can also finish its job. Hence, all processes can finish, implying that the system remains in an H-safe state. As a result, the request can be safely granted. In conclusion, the following has occurred in this example: (a) Request[i][j] $\leq$ Need[i][j] for all j at Step 1, (b) Request[i][j] $\leq$ Available[j] for all j at Step 2, and (c)

Need[i][j] $\leq$ Work[j] for all i and for all j at the first iteration of Step 5, enabling PBA to finish in three clock cycles, resulting in $O(1)$ run-time. ■

5.3.3 Proof of the Correctness of PBA

Theorem 9 *PBA always finds an H-safe sequence if and only if a system is in an H-safe state.*

Proof: We first consider the case where a system is in an H-safe state. We need to prove that PBA finds an H-safe sequence.

By Theorem 1 in Chapter 1, if a system is in an H-safe state, there must exist an H-safe sequence. Let such an H-safe sequence be $p_1, p_2, p_3, \cdots, p_{n-1}, p_n$. That is, in the H-safe state, there exists a sequence such that Need[1][j] $\leq$ Available[j] for all j, Need[2][j] $\leq$ Available[j] + Allocation[1][j] for all j, Need[3][j] $\leq$ Available[j] + Allocation[1][j] + Allocation[2][j] for all j, $\cdots$, and Need[n][j] $\leq$ Available[j] + $\Sigma_{i=1}^{i=n-1}$ Allocation[i][j] for all j.

We are considering the case where a system is in such an H-safe state. At the first step in the corresponding H-safe sequence, we already know that p_1 can proceed; thus, at the first iteration of PBA, PBA identifies p_1 as an *able-to-finish* process by comparing and ensuring Need[1][j] $\leq$ Work[j]$_0$ (i.e., Available[j]) for all j where Work[j]$_k$ denotes the value of Work[j] at k^{th} iteration. Thus, Finish[1] is set to *true* and Work[j]$_1$ (i.e., Work[j] after the first iteration) becomes Work[j]$_0$ + Allocation[1][j] for all j.

At the second iteration of PBA, PBA identifies p_2 as an *able-to-finish* process by comparing and ensuring Need[2][j] $\leq$ Work[j]$_1$ for all j; thus, Finish[2] is set to *true* and Work[j]$_2$ becomes Work[j]$_1$ + Allocation[2][j] for all j.

In the same way, PBA identifies from p_3 to p_{n-1} as *able-to-finish* processes. Finally, PBA will identify p_n as an *able-to-finish* process by comparing and ensuring Need[n][j] $\leq$ Work[j]$_{n-1}$ for all j; thus, Finish[n] will become *true* and Work[j]$_n$ will become Work[j]$_{n-1}$ + Allocation[n][j] for all j. That is, PBA will find an H-safe sequence of $p_1, \cdots, p_n$.

As a result, if a system is in an H-safe state, PBA finds an H-safe sequence.

Conversely, if PBA finds an H-safe sequence, then, by Theorem 1, the system is in an H-safe state. ■

5.3.4 Proof of the Run-time Complexity of the PBAU

Theorem 10 *PBA, when implemented in parallel hardware, completes its computation in at most n steps = $O(n)$, where n is the number of processes.*

Proof: Let us first consider Steps 1-4 and Step 6 of PBA. Since Steps 1-4 and Step 6 execute only once in PBA, these are considered to take a constant amount of time, contributing O(1) in the calculation of run-time complexity.

Let us now consider Step 5 of PBA. At each iteration of Step 5, there are three possible cases: (i) no process can proceed, (ii) only one process can proceed, and (iii) multiple processes can proceed. In case (i), PBA stops iterating Step 5. In case (ii), PBA will detect and declare only one process to be *able-to-finish* at the current iteration, thereby excluding one process from further iterations. In case (iii), if multiple processes can proceed, at one iteration PBA will declare all such processes to be *able-to-finish*, excluding multiple processes from further iterations, leaving much fewer processes than case (ii).

Therefore, case (ii) where only one process can proceed at each iteration will be the worst-case. Thus, we want to construct the maximum sized sequence where each Step 5 iteration results in case (ii). Let such a sequence be $p_1, p_2, p_3, \cdots, p_{n-1}, p_n$. Then, by construction, at the first iteration PBA identifies p_1 as the only process *able-to-finish*. At the second iteration, PBA identifies p_2 as the only process *able-to-finish*. In the same way, PBA identifies from p_3 to p_n as *able-to-finish* processes at each successive iteration. As a result, the total number of iterations of finding such a unique sequence becomes n, i.e., an $O(n)$ run-time complexity. ■

5.3.5 Comment on Livelock Avoidance in PBA

Let us consider PBA (Algorithm 7). In Line 25 of PBA, when a process requests resource(s) and the request is denied, the process would probably again request the same resource(s) some time later. Let such an interval be time t_w. If the same request(s) are repeatedly denied for a long period of time, the process could be involved in a livelock situation. To be able to break such potential livelock, PBA prepares a counter and a threshold number of repeated denials of requests can be set to detect such a potential livelock situation. Such a situation and a possible solution are given in the following example.

Example 27 Consider a system in a state that is safe but two or more processes compete for the same type of resource such that they repeat requests but the requests are not granted because resources are available but are not enough to fulfill either of the requests. In such a situation, Wait_count[i] can be used to break the livelock associated with deadlock avoidance as follows.

Let such competing processes p_1 and p_2 and assume that they are competing for resources q_1 and q_2. Whenever requests are denied, PBA performs the following. PBA increment Wait_count[1] by one each time p_1 requests but fails, and PBA increment Wait_count[2] by one each time p_2 requests but fails. Now, if either Wait_count[1] or Wait_count[2] passes some threshold that is set in advance, the corresponding process will be able to be informed of livelock, so that the process could take appropriate action (such as releasing all held resources) in such a livelock case. ■

Please note that [19] describes a similar way to resolve livelock associated with deadlock avoidance in the use of the Banker's Algorithm.

5.4 Implementation

Now we describe implementation details including architecture, circuit and equations of the PBAU.

5.4.1 Architecture of the PBAU

Figure 37 illustrates PBAU's architecture. PBAU is composed of element cells, process cells, resource cells and a safety cell in addition to a Finite State Machine (FSM) and a

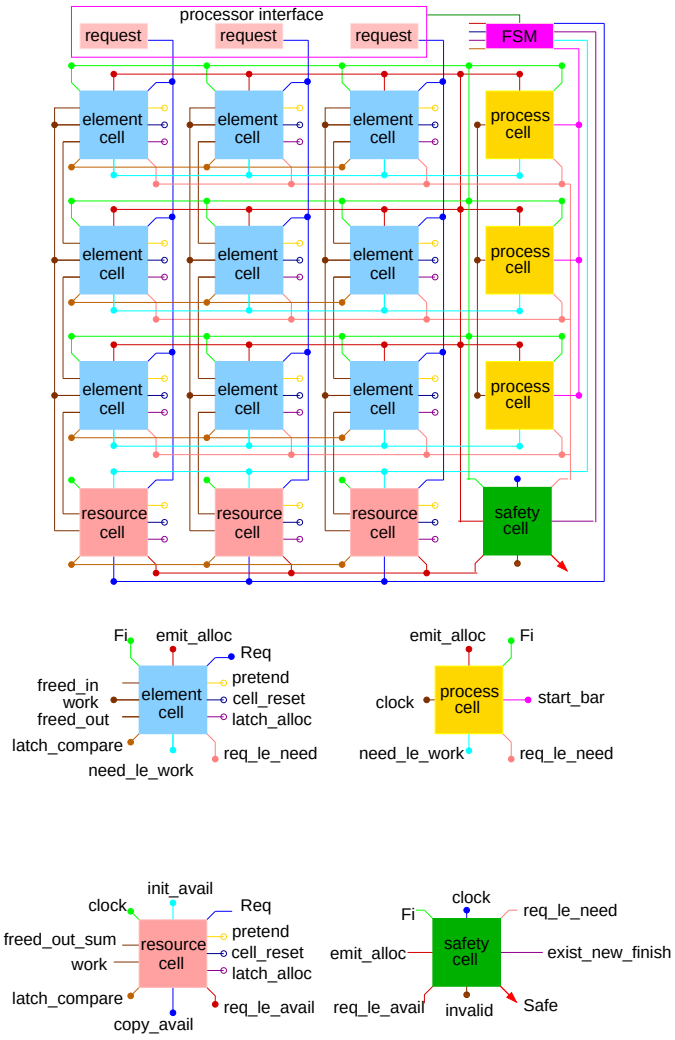


Figure 37: PBAU architecture.

processor interface.

The Processor Interface (PI) consists of command registers and status registers. PI receives and interprets commands (requests or releases) from processes as well as accomplishes simple jobs such as setting up the numbers of maximum claims and available resources as well as adjusting the numbers of allocated and available resources in the response to a release of resources. PI also returns processing results back to PEs via status registers as well as activates the FSM in response to a request for resources from a process. In the next subsection, we will describe in detail each cell in Figure 37.

5.4.2 Circuitry and Equations of the PBAU

5.4.2.1 Resource Cell

A Resource Cell (RC) is shown in Figure 38. Inputs of RC are *Data_in*, *pretend/*$\overline{restore}$, *ORed_latch_compare*, *ORed_latch_alloc_j*, *initialize_available*, *freed_out_sum_j*, *clock* and *copy_available*. Outputs are *req_le_avail_j* and Work[j].

Each system resource available for allocation has a corresponding hardware RC in the PBAU. During an initialization phase required prior to execution of PBA, the maximum number of available resource instances needs to be set in each RC. To accomplish this, for a particular RC, the total number of resource instances is sent via *Data_in* (in our current implementation, each resource has at most 16 instances, and thus *Data_in* is a 4-bit wire). The selection of *Data_in* is controlled by *pretend/*$\overline{restore}$ for possible writing to the Available Register in Figure 38. The actual latching of an input into the Available Register is controlled by either *ORed_latch_alloc_j* or *initialize_available*; in the specific case where we are in an initialization phase prior to execution of PBA, the Available Register would have been reset to zero, *pretend/*$\overline{restore}$ would be '1' thus selecting *Data_in* plus zero equals the total number of resource instances, and *initialize_available* would be used to latch in the total number of resource instances into the Available Register (please note that *ORed_latch_alloc_j* remains a zero throughout the initialization phase prior to execution of PBA). The Work Register is initialized with the value of the Available Register at a rising clock when *copy_available* is '1'.

Outputs of RC are Work[j] and a comparison result *req_le_avail_j* (i.e., the availability of a resource, Line 5 of PBA). RC has an Available Register that stores the number of instances of the resource. RC also has a Work Register[j] that temporarily stores the number of resources in the Available Register (as shown in Step 4 of PBA) plus resources to be released (i.e., *freed_out_sum_j* input) while *copy_available* is '0' by *able-to-finish* processes during iterations of Step 5. RC also has a comparator that compares Request[i][j] (assuming *Data_in* = Request[i][j]) with the Available Register (Step 2 of Algorithm 7, PBA), the

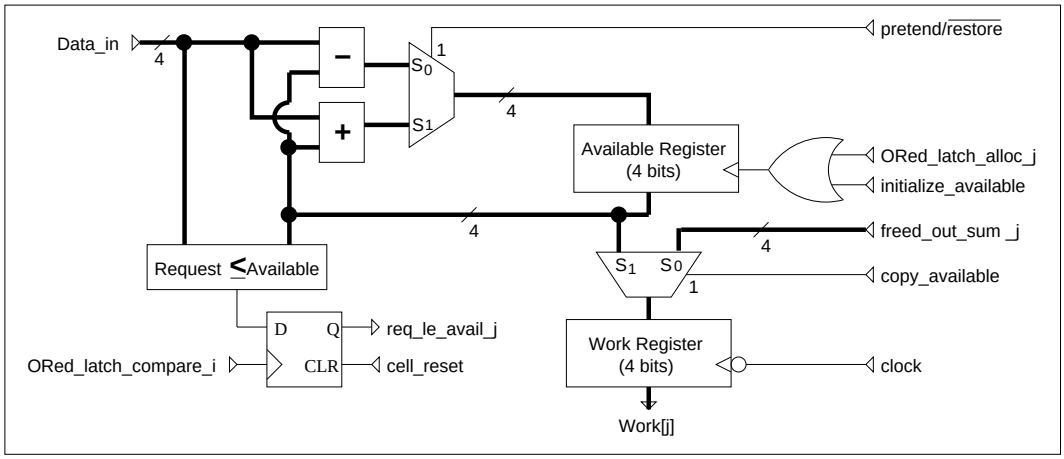


Figure 38: Logic diagram of a Resource Cell (RC).

result of which is stored into a register. A *pretend/*$\overline{restore}$ signal is also used to increase or decrease the number in the Available Register when the resource is requested or released.

The following equations represent mathematical expressions being calculated in each RC. Equation 42 corresponds to Line 5 of PBA.

$$R_{LE}V_j \Leftarrow R_{ij} \leq V_j \tag{42}$$

where $R_{LE}V_j$ denotes *req_le_avail_j*. Please see Table 11 in Section 5.3.2 for definitions of R_{ij}, V_j, etc. Equation 43 corresponds to Line 9 of PBA, which is the operation of updating available for each resource cell when some instances of a resource are allocated or released.

$$V_{j_{update}} = \begin{cases} V_j + R_{ij}, & \text{if pretend} = 1 \\ V_j - R_{ij}, & \text{if restore} = 1 \end{cases} \tag{43}$$

Equation 44 corresponds to Line 17 of PBA.

$$Work_{j_{k+1}} = Work_{j_k} + \Sigma_i freed\_out_{ij_k} \text{ (i.e., } freed\_out\_sum\_j) \tag{44}$$

where k refers to k^{th} iteration, and $k+1$ refers to $(k+1)^{\text{th}}$ iteration.

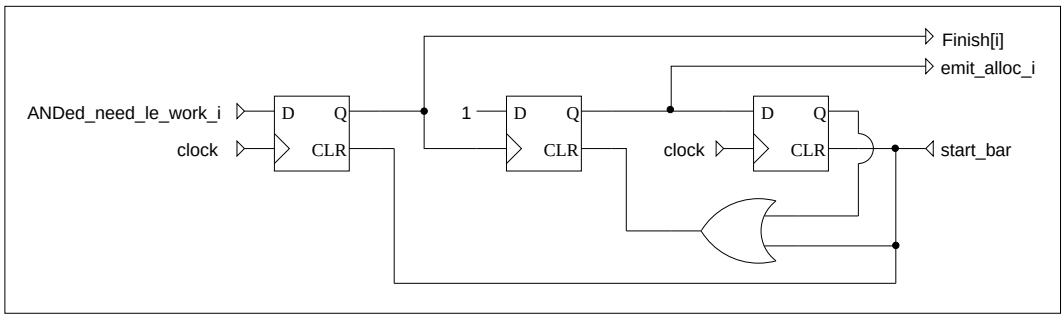


Figure 39: Logic diagram of a Process Cell (PC).

5.4.2.2 Process Cell

A Process Cell (PC) is shown in Figure 39. Inputs of PC are *ANDed_need_le_work_i* and *start_bar*, while outputs are Finish[i] and *emit_alloc_i*. *start_bar* is a start signal of an H-safety check. *ANDed_need_le_work_i* is the result of an n-bit AND of the signals of all the comparison results of Request[i][j] $\leq$ Work[j] for all j from all element cells corresponding to a process (Line 15 of PBA). PC generates Finish[i] for process i and issues an addition signal (i.e., *emit_alloc_i*) that makes allocated resources to this process available to later processes in an H-safe sequence.

The following equations represent mathematical expressions being calculated in each PC. Equation 45 corresponds to Line 15 of PBA.

$$\begin{gathered} N_{LE}W_i = AND\Sigma_j \; N_{LE}W_{ij} \\ (AND\Sigma_j \text{ means AND for all } j) \end{gathered} \tag{45}$$

where $N_{LE}W_i$ denotes *ANDed_need_le_work_i*.

Equations 46 and 47 correspond to Line 18 of PBA.

$$F_i \Leftarrow N_{LE}W_i \; (\Leftarrow \text{means } clock\ synchronized) \tag{46}$$

$$emit\_alloc_i \Leftarrow F_i \tag{47}$$

5.4.2.3 Element Cell

An Element Cell (EC) is shown in Figure 40. Inputs of EC are Work[j], *initialize_max_clk_i*, *Data_in*, *pretend/*$\overline{restore}$, *emit_alloc_i*, *latch_alloc_clk_i* and *freed_in_ij*. Outputs of EC are

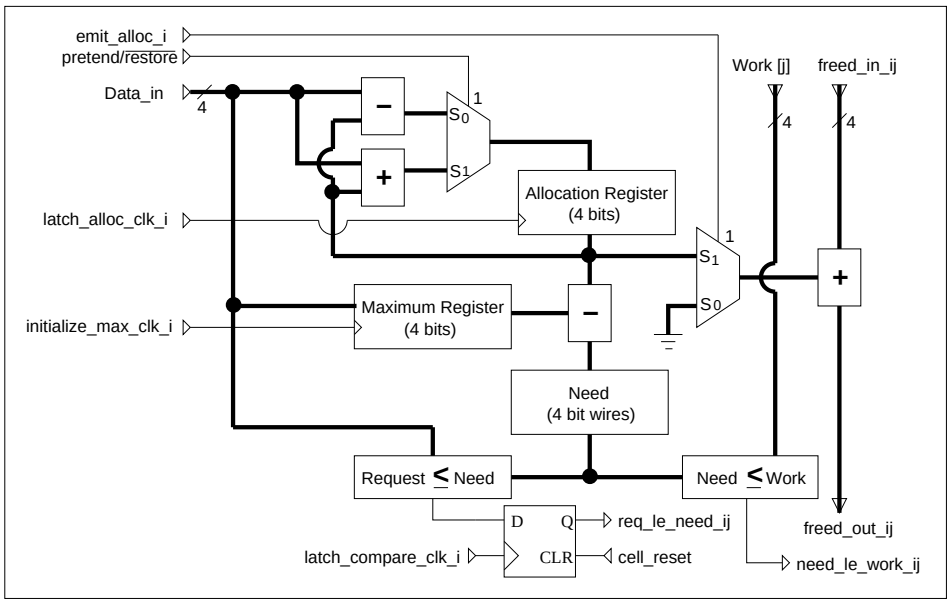


Figure 40: Logic diagram of an Element Cell (EC).

req_le_need_ij, *need_le_work_ij* and *freed_out_ij*.

Each EC stores the number of resource instances (e.g., resource j) allocated for a particular process (e.g., process i) as well as the maximum claim of resource instances for a specific process. Obviously, the hardware will include an EC array where there is one EC per resource j, process i pair. During an initialization phase required prior to execution of PBA, the maximum number of resource claims for each specific process needs to be set in each EC. To perform this operation, for a particular EC, the maximum claim of resource instances is sent via *Data_in*. The latching of *Data_in* into the Maximum Register is controlled by *initialize_max_clk_i*.

In Step 3 of PBA, each allocation amount of resource instances is sent via *Data_in*. The selection of *Data_in* is controlled by *pretend/*$\overline{restore}$ for possible writing to the Allocation Register in Figure 40. The actual latching of an input into the Allocation Register is controlled by *latch_alloc_clk_i*; in the specific case where we are in an initialization phase prior to execution of PBA, the Allocation Register is reset to zero. When there exists a change in allocation due to a grant event, *pretend/*$\overline{restore}$ is set to a '1', selecting *Data_in*,

which will be added to the value of resource instances currently in the Allocation Register. However, when some instances are being de-allocated due to a release event, by setting *pretend/*$\overline{restore}$ to zero, an amount in *Data_in* will be subtracted from a current amount in the Allocation Register

EC performs two comparisons: Request[i][j] $\leq$ Need[i][j] and Need[i][j] $\leq$ Work[j]. The former comparison result (i.e., Request[i][j] $\leq$ Need[i][j]) is stored into a one-bit register and then sent via *req_le_need_ij* while the latter comparison result (i.e., Need[i][j] $\leq$ Work[j]) is directly sent via *need_le_work_ij*.

EC emits the value of the Allocation Register to *freed_out_ij*, which is controlled by *emit_alloc_i* when EC belongs to an *able-to-finish* process (i.e., Need[i][j] $\leq$ Work[j] for all j). However, if EC does not belong to an *able-to-finish* process, *emit_alloc_i* will be zero; thus *freed_out_ij* will just contain the value of input *freed_in_ij*.

In addition, there are two muxes, two subtracters and two adders. One adder is used to increase the number of allocation instances of the requested resource, and one subtracter is used to restore the temporarily increased number of instances if the safety test fails (see Example 25 for a sample execution of the safety test). Another subtracter is used to calculate the equation Need[i][j] = Maximum[i][j] – Allocation[i][j]. The other adder is used to make allocated instances (to this cell) available to later processes in an H-safe sequence.

The following equations represent mathematical expressions being calculated in each EC. Equation 48 corresponds to Line 2 of PBA.

$$R_{LE}N_{ij} \Leftarrow R_{ij} \leq N_{ij} (\Leftarrow \text{means } clock\ synchronized) \tag{48}$$

where $R_{LE}N_{ij}$ denotes *req_le_need_ij*. Please see Table 11 in Section 5.3.2 for definitions of R_{ij}, N_j, X_{ij}, G_{ij} and W_j.

While Equation 49 corresponds to Lines 10 and 25 of PBA (updating allocation), Equation 50 corresponds to Line 11 (updating Need[i][j] for each cell), and Equation 51 corresponds to Line 15.

$$G_{ij_{new}} = \begin{cases} G_{ij} + R_{ij}, & \text{if pretend} = 1 \\ G_{ij} - R_{ij}, & \text{if restore} = 1 \end{cases} \tag{49}$$

$$N_{ij} = X_{ij} - G_{ij} \tag{50}$$

$$N_{LE}W_{ij} = 1, \text{ if } N_{ij} \leq W_j \tag{51}$$

where $N_{LE}W_{ij}$ denotes *need_le_work_ij*.

Equation 52 corresponds to Line 17 of PBA, which are the connections and operation of adding resources in Available[] and to be potentially available.

$$\begin{aligned} & freed\_in_{1j} = Work_j \\ & freed\_out_{ij} = freed\_in_{i,j} + freed\_alloc_{ij} \text{ for } 1 \leq i \leq n \\ & freed\_in_{ij} = freed\_out_{i-1,j} \text{ for } 2 \leq i \leq n \\ & freed\_out\_sum_j = freed\_out_{nj} \end{aligned} \tag{52}$$

5.4.2.4 Safety Cell

A Safety Cell (SC) is shown in Figure 41. Inputs are *ANDed_req_le_avail*, n number of *ANDed_req_le_need* signals, *emit_alloc_i* and Finish[i]. Outputs are *Safe*, *invalid* and *exist_new_finish*. *ANDed_req_le_avail* is the result of an m-bit AND of the comparisons of Request[i][j] $\leq$ Available[j]. Each individual *ANDed_req_le_need* bit signal is the result of an m-bit AND of comparisons of Request[i][j] $\leq$ Need[i][j] for each i for all j. SC inputs comparison results (i.e., *ANDed_req_le_avail*, n number of *ANDed_req_le_need* signals) from all resource cells as well as all element cells and generates the H-safety result (i.e., *Safe*). Thus, SC checks if Request[i][] is valid and stores the result into a register, the output of which is *invalid*, which is sent out at the rising edge of *check_valid_clk*. SC

also checks whether or not there exist more *able-to-finish* rows (i.e, *exist_new_finish*). If no more *able-to-finish* rows exist, iteration stops, and the safety result is decided by examining all Finish[] coming from all process cells.

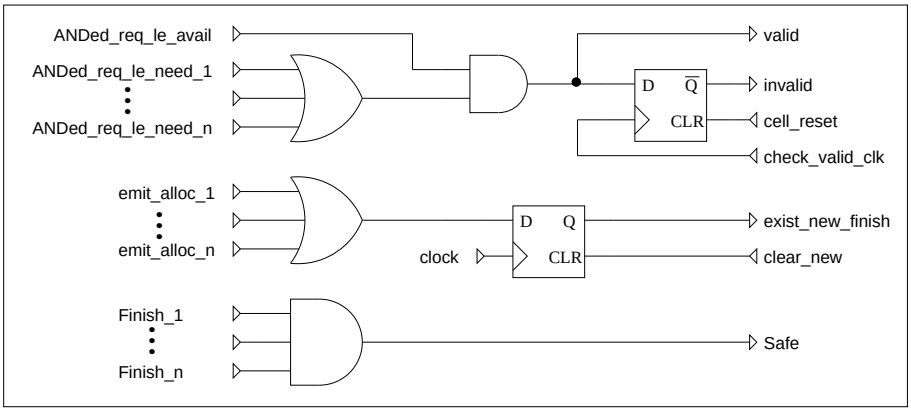


Figure 41: Logic diagram of a Safety Cell (SC).

The following equations represent mathematical expressions being calculated in each SC. Equation 53 corresponds to Lines 2 and 5 of PBA.

$$Valid = (AND\Sigma_j\ R_{LE}V_j) \cdot (OR\Sigma_i\ AND\Sigma_j\ R_{LE}N_{ij}) \tag{53}$$

Equation 54 corresponds to Line 16 of PBA.

$$\exists F_{new} = OR\Sigma_i\ emit\_alloc_i \tag{54}$$

Finally, Equation 55 corresponds to Lines 22 and 23 of PBA, producing the safety result.

$$safe = AND\Sigma_i\ F_i \tag{55}$$

Now, we will give a specific operation example of the PBAU.

Example 28 Brief operation of cells

Let us reconsider Example 26 and focus on element cells e_{11} and e_{12} (corresponding to p_1) and resource cells c_1 (corresponding to q_1) and c_2 (corresponding to q_2). Assuming that the current allocation state is as shown in Table 17, internal values of e_{11} will be Maximum[1][1] = 3, Allocation[1][1] = 1, and thus Need[1][1] = 2. Internal values of e_{12} will be Maximum[1][2] = 2, Allocation[1][2] = 1, and thus Need[1][2] = 1. In addition, Available[1] of c_1 will be 3, and Available[2] of c_2

will also be 3. Now if p_1 requests one instance of q_1 and one instance of q_2, both Request[1][1] and Request[1][2] become 1. Then, since Request[1][1] $\leq$ Need[1][1], *req_le_need_11* (see Figure 40) becomes 1; and since Request[1][2] $\leq$ Need[1][2], *req_le_need_12* becomes 1; these results indicate that Step 1 of Algorithm 7 is satisfied.

At Step 2, since Request[1][1] $\leq$ Available[1] in resource cell c_1, *req_le_avail_1* (see Figure 38) becomes 1; and since Request[1][2] $\leq$ Available[2] in resource cell c_2, *req_le_avail_2* becomes 1; these results represent that Step 2 is also satisfied.

After this, at Step 3, when grant is pretended as shown in Table 18, $pretend/\overline{restore}$ in Figures 40 and 38 becomes 1, and by signal *latch_alloc_clk_1*, the internal values are updated as follows. For e_{11}, Allocation[1][1] = 2 and Need[1][1] = 1. For c_1, Available[1] = 2. For e_{12}, Allocation[1][2] = 2 and Need[1][2] = 0. For c_2, Available[2] = 2.

Then, at Step 4, c_1 emits Work[1] (i.e., Available[1]), and c_2 emits Work[2] (i.e., Available[2]). Finally, at Step 5, to find processes *able-to-finish*, comparisons (i.e., for all j, Need[1][j] $\leq$ Work[j]) are performed in element cells e_{11} and e_{12}. Since Need[1][1] $\leq$ Work[1] and Need[1][2] $\leq$ Work[2], p_1 becomes *able-to-finish*, i.e., p_1 can finish its job.

Similarly for the rest of i, Need[i][j] $\leq$ Work[j] for all j are performed at the same time in parallel. As explained in Example 26, all rows (i.e., processes) become *able-to-finish*; thus, the system remains H-safe. ■

5.4.2.5 Finite State Machine (FSM)

Figure 42 illustrates the transition diagram of the PBAU FSM along with input and output signals. The FSM sequences PBA execution (i.e., Algorithm 7). If the FSM receives a start signal to initiate PBA execution, the FSM issues a *cell_reset* signal that resets all internal registers of all cells at the first clock. At the second clock, the FSM checks if Request[i][] is valid (corresponding to Steps 1 and 2 in Algorithm 7); i.e., in all resource cells (all RC hardware units), a set of comparisons between Request[i][j] and Available[j] for all j is carried out, and another set of comparisons between Request[i][j] and Need[i][j] for all j is carried out in all element cells in a specific row corresponding to the requester (i.e., process i). Please note that the correspondences between rows and columns with processes and resources differ from the correspondences in the DDU in Chapter 3. If the

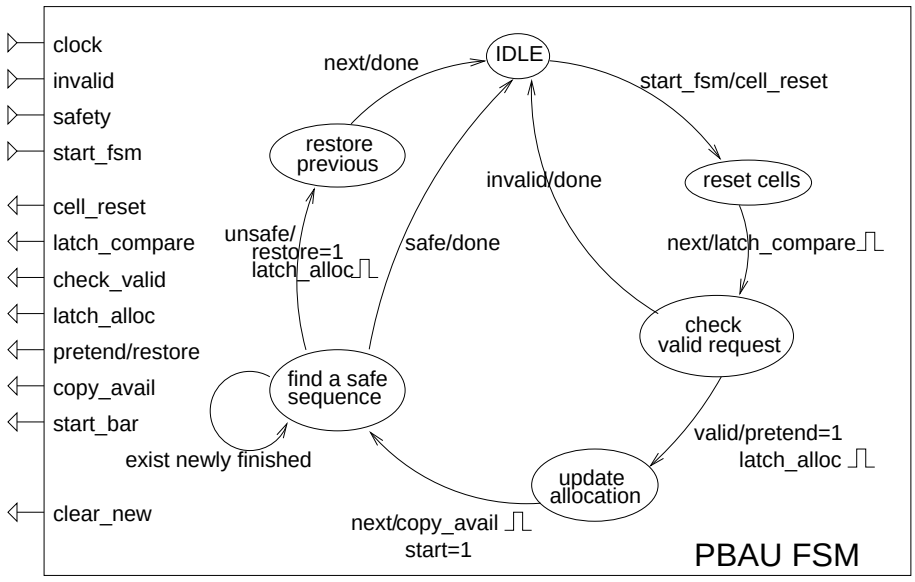


Figure 42: Finite state machine of the PBAU.

command is found to be valid, then Request[i][] is assumed to be acceptable, and at the third clock, the FSM updates Available[] and Allocation[i][] according to the numbers of instances of all requested resources in parallel. Please note that at this moment Need[i][j] (i.e., Maximum[i][j] – Allocation[i][j]) is calculated automatically inside each element cell. Within the third clock cycle, Available[] is also copied to Work[] (corresponding to Step 4). Please note that all Finish[] are set to false (stated in Step 4) by the reset of cells. At the fourth clock, the iterations of Step 5 begin. For all cells in rows for which Finish[i] is *false*, PBA checks if Need[i][j] $\leq$ Work[j] for all j in parallel throughout the whole matrix, which helps reduce the run-time complexity. At the fifth clock, if there exists row(s) such that all element cells in a row satisfy the equation (Need[i][j] $\leq$ Work[j] for all i and j) (i.e., the process corresponding to the row can finish its job by acquiring some or all of resources currently available), such row(s) are set to be *able-to-finish* and their resources are added to Work[] (Line 17 in Algorithm 7). If such row(s) exist, then after excluding such row(s), Step 5 repeats until no more such row(s) exist. Please note that in cases where all rows satisfy Need[i][j] $\leq$ Work[j] for all i and j, PBA finishes at the fifth clock cycle, resulting in an $O(1)$ run-time. Since each iteration requires one clock cycle, in the worst

case where there exists only one unique H-safe sequence, the number of iterations will be n clock cycles, where n is the number of processes in the system.

Iterations cease to end if no more *able-to-finish* rows exist. Then, by checking all Finish[], the safety decision is made, and if it is found to be H-unsafe, the FSM restores pretended Allocation[i][], Need[i][] and Available[] by issuing a restore signal to all resource cells and element cells in the row corresponding to the requester.

5.4.3 Synthesized Result of the PBAU

We use the Synopsys Design Compiler [52] to synthesize various PBAU sizes with the QualCore Logic .25μm standard cell library [39]. We synthesize with a clock period of 4 ns (250 MHz). The synthesis result is shown in Table 19. The "Area" column denotes the area in units equivalent to a minimum-sized two-input NAND gate in the library. PBAU5x5 represents a PBAU for five processes and five resources (each resource can have up to 16 instances). In a case where an SoC contains five PowerPC 755 PEs (1.7M gates each) and 16MB memory (33.5M gates), the resulting MPSoC area, the sum of the areas of 16MB of memory plus five MPC755's plus PBAU20x20 (i.e., 33.5M + 1.7M $\times$ 5 + 19753), is 42019753 gates. Thus, the area overhead in the SoC due to the PBAU 20x20, i.e., the area of PBAU20x20 divided by the total MPSoC area is approximately .05% (i.e., 19753/42019753).

Table 19: Synthesized result of the PBAU.

Synthesis Result	PBAU5x5	8x8	10x10	15x15	20x20
Area (w.r.t. 2-input NAND)	1303	3243	5030	11158	19753
Number of lines of Verilog	600	700	770	1000	1350

5.4.4 Run-time Complexity of the PBAU

The run-time complexity of a generic implementation of the traditional BA in software is $O(m \times n^2)$, where m and n are the numbers of resources and processes, respectively [31].

By implementing PBA in hardware able to exploit full parallelism, we achieve a run-time of $O(1)$ in the best case (i.e., the cases of system states where for all i and for all j, Need[i][j] $\leq$ Available[j]), $O(n)$ in the worst case (i.e., the cases where there exists only one unique H-safe sequence of one by one increment order of *able-to-finish* processes), and seems to be n/2 clock cycles on average (in our experiments).

Let us illustrate how we achieve such a run-time complexity from Algorithm 7. Steps 1 and 2 of Algorithm 7 can execute in parallel in one clock cycle, and if Request[i][] is permissible, then Step 3 (pretending to allocate the requested resources) and Step 4 (iteration preparation for the H-safety check) can be done in one clock in parallel. Then, each iteration of Step 5 takes one clock until all rows (i.e., processes) become *able-to-finish*. Since in the worst case, only one after one process can finish, the worst case number of iterations consumes n clock cycles for Step 5, where n is the number of processes in the system.

5.5 Experiments

5.5.1 Simulation Environment Setup for PBAU evaluation

The experimental simulations were carried out using Seamless Co-Verification Environment (CVE [33]) aided by Synopsys VCS [53] for Verilog HDL simulation and XRAY [34] for software debugging. We use Atalanta RTOS version 0.3 [51], a shared-memory multiprocessor RTOS. The other simulation setups not mentioned here such as a bus clock rate and a system memory size are the same in Section 3.4.1.

5.5.2 Experimental System

For the experiment, we simulate an MPSoC with five Motorola MPC755s and resources similar to Figure 36. Each MPC755 has separate instruction and data L1 caches each of size 32KB. The MPSoC also has the following three types of resources: an SoCDMMU [46] with 10 blocks of allocable memory (q_1), a counting semaphore with a group of five DSP processors (q_2) and another counting semaphore with seven I/O buffers (q_3). These three types of resources have timers, interrupt generators and input/output ports as

needed to operate properly in the MPSoC. In addition, the MPSoC has a PBAU for five processes and five resources, an arbiter and 16MB of shared memory including the allocable memory. The master clock rate of the bus system is 10 ns. Code for each MPC755 runs on an instruction-accurate (not cycle-accurate) MPC755 simulator provided by Seamless CVE [33]. Everything else other than the MPC755s are described in Verilog HDL and simulated in Synopsys VCS [53]. We invoke processes $p_1, \cdots, p_5$ on PE1, $\cdots$, PE5, respectively.

5.5.3 Application Example

We execute a sample robotic application which performs the following: recognizing objects, avoiding obstacles and displaying trajectory requiring DSP processing; robot motion and data recording involving accessing IO buffers; and proper real-time operation (e.g., maintaining balance) of the robot demanding fast and deterministic allocation and deallocation of memory blocks. This application invokes a sequence of requests and releases. The sequence has ten requests, six releases and five claim settings with one request that violates a pre-declared maximum claim (e.g., Request[i][j] $>$ Need[i][j]) and one additional request that leads to an H-unsafe state as shown in Table 20. Please note that every command is processed by an avoidance algorithm (either PBAU or BA in software). Recall that there is no constraint on the ordering of resource usage.

Detailed sequence explanation is as follows. There are five processes and three resources in the system. Table 21 shows the available resources and maximum claims of each process in the system at time t_5 (*Maximum* equals *Need* currently).

Table 22 shows the resource allocation state at time t_{10} as processes are using resources.

After two more requests, Table 23 shows the resource allocation state at time t_{12}.

So far, all requests result in H-safe states. However, at time t_{13}, when p_5 requests one additional instance of resource q_1, the system results in an H-unsafe state if the request is granted. Thus, PBAU rejects the request; the wait count (please see Table 11) for p_5 is

Table 20: A sequence of requests and releases for PBAU test.

Time	Events
t_0	The application starts, and the numbers of available resources in the system are set.
t_1	p_1 sets its maximum claims for each resource.
t_2	p_2 sets its maximum claims for each resource.
t_3	p_3 sets its maximum claims for each resource.
t_4	p_4 sets its maximum claims for each resource.
t_5	p_5 sets its maximum claims for each resource (see Table 21).
t_6	p_1 requests one instance of q_2.
t_7	p_2 requests two instances of q_1.
t_8	p_3 requests three instances of q_1 and two instances of q_3.
t_9	p_4 requests two instances of q_1, one instance of q_2 and one instance of q_3.
t_{10}	p_5 requests two instances of q_3 (Table 22).
t_{11}	p_1 requests two instances of q_2 and one instance of q_3.
t_{12}	p_5 requests one instance of q_1. So far, all requests make the system remain H-safe. (Table 23).
t_{13}	p_5 again requests one more instance of q_1, which results in H-unsafe. Thus, this request is denied. The wait count for p_5 is increased.
t_{14}	p_3 releases two instances of q_1 and two instances of q_3 (Table 24).
t_{15}	p_3 initiates a false request (i.e., it requests five instances of q_1, q_2 and q_3, respectively), which of course is denied.
t_{16}	p_5 again requests one more instance of q_1, which now results in H-safe. Thus, this request is granted (Table 25). The wait count for p_5 is cleared.
t_{17}	p_1 finishes its job and releases three instances of q_2 and one instance of q_3.
t_{18}	p_2 releases two instances of q_1.
t_{19}	p_3 releases one instance of q_1.
t_{20}	p_4 releases two instances of q_1, one instance of q_2 and one instance of q_3.
t_{21}	p_5 releases two instances of q_1 and two instances of q_3, the application ends.

increased, and p_5 needs to rerequest q_1 later.

At time t_{14}, p_3 releases two instances of q_1 and two instances of q_3, and the allocation state is shown in Table 24.

At time t_{16}, p_5 rerequests one additional instance of resource q_1, and the request is granted as shown Table 25. The wait count for p_5 is cleared.

After time t_{16}, as time progresses, all processes finish their jobs and release allocated resources.

Table 21: Initial resource allocation state at time t_5.

	Maximum	Allocation	Need	Available
	q_1 q_2 q_3	q_1 q_2 q_3	q_1 q_2 q_3	q_1 q_2 q_3
p_1	7 5 3	0 0 0	7 5 3	10 5 7
p_2	3 2 2	0 0 0	3 2 2	
p_3	9 0 2	0 0 0	9 0 2	
p_4	2 2 2	0 0 0	2 2 2	
p_5	4 3 3	0 0 0	4 3 3	

Table 22: Resource allocation state at time t_{10}.

	Allocation	Need	Available
	q_1 q_2 q_3	q_1 q_2 q_3	q_1 q_2 q_3
p_1	0 1 0	7 4 3	3 3 2
p_2	2 0 0	1 2 2	
p_3	3 0 2	6 0 0	
p_4	2 1 1	0 1 1	
p_5	0 0 2	4 3 1	

Table 23: Resource allocation state at time t_{12}.

	Allocation	Need	Available
	q_1 q_2 q_3	q_1 q_2 q_3	q_1 q_2 q_3
p_1	0 3 1	7 2 2	2 1 1
p_2	2 0 0	1 2 2	
p_3	3 0 2	6 0 0	
p_4	2 1 1	0 1 1	
p_5	1 0 2	3 3 1	

With the above scenario, summarized in Tables 20-25, we measure two figures, the average execution time of the deadlock avoidance algorithm used and the total execution time of the application in two cases: (i) using PBAU versus (ii) using the Banker's Algorithm in software.

5.5.4 Experimental Result

Table 26 shows that PBAU achieves about a 1600X speedup of the average algorithm execution time and gives a 19% speedup of application execution time over avoiding deadlock

Table 24: Resource allocation state at time t_{14}.

	Allocation			Need			Available		
	q_1	q_2	q_3	q_1	q_2	q_3	q_1	q_2	q_3
p_1	0	3	1	7	2	2	4	1	3
p_2	2	0	0	1	2	2			
p_3	1	0	0	8	0	2			
p_4	2	1	1	0	1	1			
p_5	1	0	2	3	3	1			

Table 25: Resource allocation state at time t_{16}.

	Allocation			Need			Available		
	q_1	q_2	q_3	q_1	q_2	q_3	q_1	q_2	q_3
p_1	0	3	1	7	2	2	3	1	3
p_2	2	0	0	1	2	2			
p_3	1	0	0	8	0	2			
p_4	2	1	1	0	1	1			
p_5	2	0	2	2	3	1			

with BA in software (the speedup is calculated according to the formula by Hennessy and Patterson [18]). Please note that during the run-time of the application, each avoidance method (PBAU or BA in software) is invoked 22 times in both cases, respectively (since every request and release invokes a deadlock avoidance calculation). Table 27 represents the average algorithm execution time distribution in terms of different types of commands.

Thus, while BA in software spends about 5400 clock cycles on average at each invocation in this experiment, PBAU only spends 3.32 clocks on average. Please note that this comparison is not exact since, as already stated in Section 3.4.2, we use an instruction accurate (not cycle accurate) MPC755 instruction-set simulator, and thus may be off by as much as an order of magnitude.

5.6 Summary

A novel Parallel Banker's Algorithm (PBA) for multiple-instance multiple-resource systems and its hardware implementation, which we call Parallel Banker's Algorithm Unit

Table 26: Application execution time comparison for PBAU test.

Method of Implementation	Algorithm Exec. Time	PBAU Speedup	Application Exec. Time	Application Speedup
PBAU (hardware)	3.32	$\frac{5398.4-3.32}{3.32} = 1625X$	185716	$\frac{221259-185716}{185716} = 19\%$
BA in software	5398.4		221259	

*The time unit is a clock cycle, and the values are averaged.

Table 27: Execution time comparison between PBAU vs. PBA in software.

Method of Implementation	Set Available	Set Max Claim	Request Command	Release Command	Wrong Command
# of commands	1	5	9	6	1
PBAU (hardware)	1	1	6.5	1	2
BA in software	416	427	11337	2270	560

*The time unit is a clock cycle, and the values are averaged if there were multiple commands of the same type. "#" denotes "the number of".

(PBAU), are described in this chapter. PBAU gives an $O(n)$ run-time complexity with the best case of $O(1)$; the result seems to be an average run-time of approximately n/2 clock cycles in most cases. PBAU provides a multiprocessor system with a very fast and low area way of avoiding deadlock at run-time, which helps free programmers from worrying about deadlock. Whenever a request occurs in a system, PBAU checks for the safety of its grant. The request is granted provided that the system can remain in an H-safe state.

We demonstrated the following through an experiment: (i) PBAU automatically avoids deadlocks as well as reduces the deadlock avoidance time by 99% *(roughly 1600X)* as compared to the Banker's Algorithm (BA) in software; and (ii) PBAU achieved in a particular example a 19% speedup of application execution time in an experiment as compared to the execution time of the same application that uses BA in software.

Finally, the MPSoC area overhead due to PBAU is small, under 0.05% in our candidate MPSoC example.

CHAPTER VI

INTEGRATING THE DDU, DAU AND PBAU INTO THE δ HW/SW RTOS PARTITIONING FRAMEWORK

6.1 Introduction

In this chapter we will briefly introduce the δ hardware/software Real-Time Operating System (RTOS) framework and then describe the methodology of the framework as well as the integration of hardware deadlock solutions discussed previously (i.e., the DDU, DAU and PBAU) into the framework. At the end, we briefly describe a separate automatic Intellectual Property (IP) generation tool for the hardware deadlock solutions.

The initial δ hardware/software RTOS/MPSoC design framework has been proposed in [26, 28, 35, 36]. As MPSoC designs become more common, hardware/software codesign engineers face new challenges involving operating system integration. The δ hardware/software RTOS/MPSoC codesign framework provides a novel methodology of hardware/software partitioning of operating systems. The δ framework is used to configure and generate simulatable RTOS/MPSoC designs having both appropriate hardware and software interfaces as well as system architecture. The δ framework is specifically designed to help RTOS/MPSoC designers very easily and quickly explore the available design space with different hardware and software modules so that they can efficiently search and discover several compact solutions matched to the specifications and requirements of their design prior to any actual implementation.

The δ framework shown in Figure 43 generates a configured RTOS/MPSoC design that is simulatable on a hardware/software cosimulation environment after the generated

design is compiled. Hardware designs are described in a Hardware Description Language (HDL) such as Verilog. Software designs could be described in any language although we have only used C in our designs. The δ framework has been developed to help RTOS/MPSoC designers explore their design space more easily and quickly with available hardware/software modules so that users can decide their critical decisions earlier in the design phase of their target product(s). We have integrated the DDU, DAU and PBAU into the δ framework [24, 37].

6.2 Methodology

The δ hardware/software RTOS generation framework for MPSoC (shown in Figure 43) was proposed to enable automatic generation of different mixes of predesigned hardware/ software RTOS components that fit the target MPSoC the user is designing. Thus, the δ framework helps a user explore which configuration is most suitable for the user's target and application or set of applications. In other words, the δ framework is specifically designed to provide a solution to rapid RTOS/MPSoC (both hardware and software) design space exploration so that the user can easily and quickly find a few optimal RTOS/MPSoC architectures that are most suitable to his or her design goals.

From the initial implementation [26, 28, 35, 36], we have extended the δ framework to include parameterized generators of hardware IP components (i.e., automatically configurable to fit a desired target architecture) as well as the generation of various types of bus systems [24, 37].

Figure 44 shows Graphical User Interface (GUI) for the δ framework version 2.0, which integrates all parameterized generators we have and generates an RTOS/MPSoC system.

Here we summarize each generator briefly. For more information, please see specific references. When a user wants to create his or her own specific bus system, the user clicks "Bus configuration" (shown at the top right of Figure 44), which brings up a pop-up window (shown in Figure 45) in which the user specifies address and data bus widths as well as

detailed bus topology for each subsystem in subsequent windows (shown in Figures 46 and 47) for a system with a hierarchical bus structure. After appropriate inputs are entered, the tool will generate a user specified bus system with a specified hierarchy. Further details

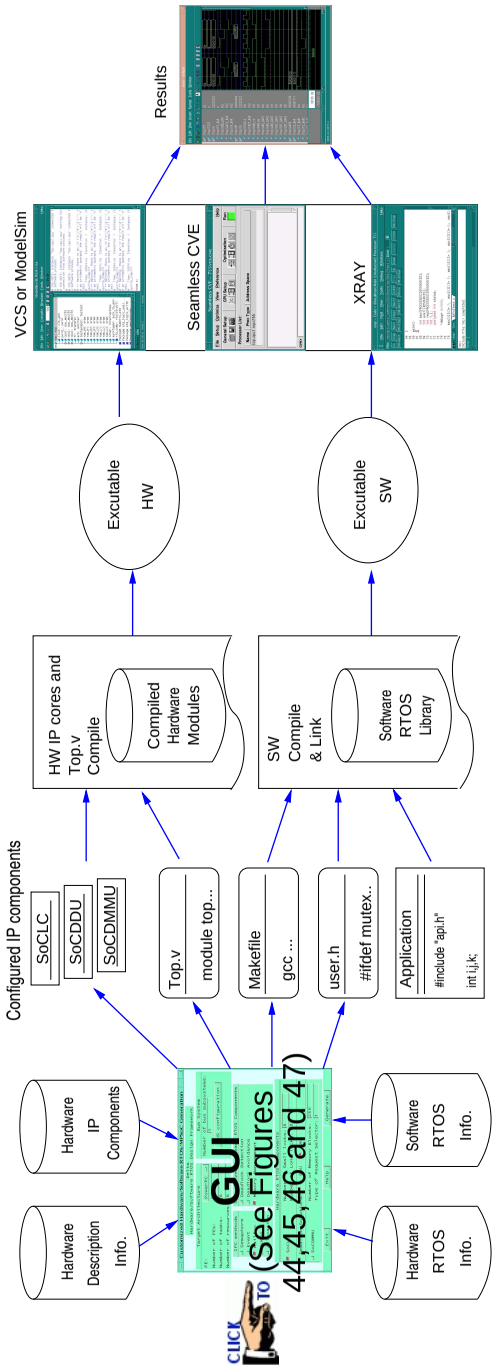


Figure 43: The δ hardware/software RTOS design framework.

about bus system generation is described in [41, 42, 43].

At the bottom of Figure 44, there are several options for “Hardware RTOS Components”: multiple deadlock detection/avoidance solutions (i.e., the Deadlock Detection Unit

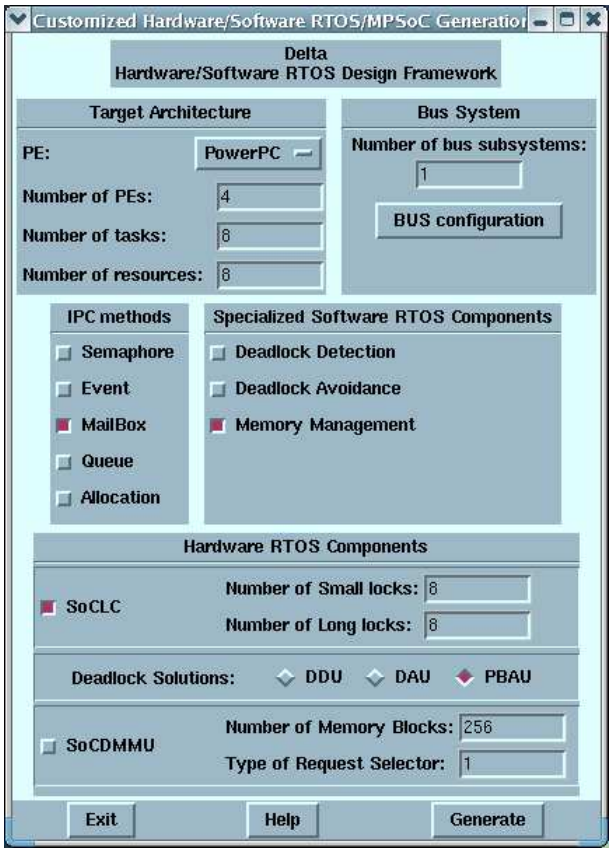


Figure 44: GUI of the δ framework.

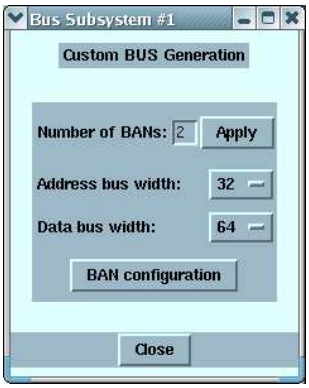


Figure 45: Bus system configuration.

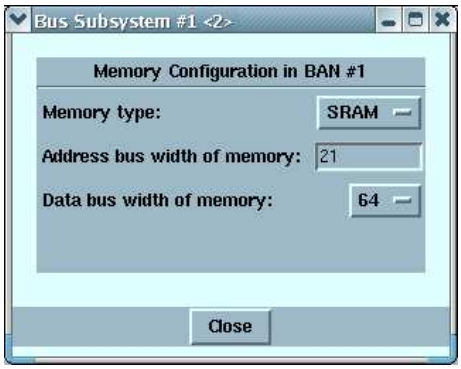


Figure 46: Bus subsystem memory configuration.

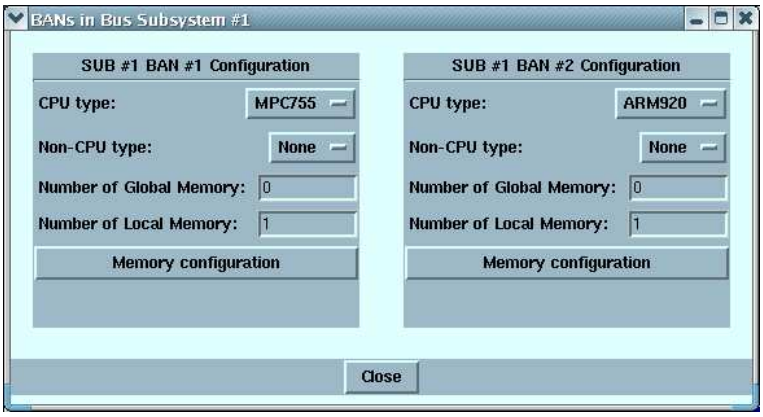


Figure 47: Bus subsystem configuration.

(DDU), the Deadlock Avoidance Unit (DDU) and the Parallel Banker's Algorithm Unit (PBAU)), the SoC Lock Cache (SoCLC [1, 2, 4]) and the SoC Dynamic Memory Management Unit (SoCDMMU [45, 46, 47]).

In addition to selecting hardware RTOS components, the δ framework version 2.0 can also manipulate the size and type of each RTOS component by use of input parameters. For instance, when the user wants to include SoCLC, he or she can also specify the number of small locks and the number of long locks (equivalent to semaphores) according to the expected requirements for his or her specific target (or goal). The detailed parameterized SoCLC generation is discussed in [1, 3].

For deadlock hardware components, after a user selects either the Deadlock Detection Unit (DDU), the Deadlock Avoidance Unit (DAU) for single-instance resource systems or PBAU for multiple-instance resource systems, the GUI tool generates a deadlock IP component with the designated type and an appropriate size according to the number of tasks and resources specified in the Target Architecture window (see upper left of Figure 44).

For the SoCDMMU IP component, the user can specify the number of memory blocks (available for dynamic allocation in the system) and other parameters, and then the GUI tool will generate a user specified SoCDMMU. The detailed parameterized SoCDMMU generation is addressed in [45, 47].

In an earlier version of the δ framework, we made the GUI generate a Verilog HDL file that describes a complete hardware system in Verilog, which was a good approach for the users who describe their hardware design in Verilog. However, since there are many VHDL users, we decided to support designs described in VHDL. In the enhanced method, we separate a HDL top file generation from the generation of component modules, and configurable modules are generated by the method described earlier. Other modules that have no necessity of configuration may be precompiled and stored in the *work* directory.

We briefly describe an HDL top file generation process in the following example.

Example 29 As shown in Figure 48, the GUI tool generates a Verilog top file according to the description of a user specified system with hardware IP components. For instance, a user selects a system having three PEs and an DDU for 10 tasks and 10 resources. The generation process starts with the DDU system description in the description library. The DDU system description lists modules necessary to build the DDU system, such as PEs, L2 memory, a memory controller, a bus arbiter, an interrupt controller and an DDU. The Verilog top file generator, which we call *Archi_gen*, writes instantiation code for each module in the list of the DDU description to a file. *Archi_gen* also includes multiple instantiation code of the same type IP with distinct identification numbers since some modules such as PEs need to be instantiated multiple times. Then, *Archi_gen* writes necessary wires described in the DDU description, and then writes initialization routines necessary to execute simulation. Later by compiling *Top.v*, a specified target hardware architecture will be ready for exploration. ■

So far, we briefly introduced the δ hardware/software RTOS framework and described the integration of hardware deadlock solutions discussed previously (i.e., the DDU, DAU and PBAU) into the framework. In the next section, we will briefly describe a separate automatic IP generation tool for such deadlock hardware solutions.

6.3 Automatic Generation of the DDU, DAU and PBAU

We believe an automated Intellectual Property (IP) tool can be developed for hardware deadlock solutions (i.e., the DDU, DAU and PBAU) . We outline the beginning of such a

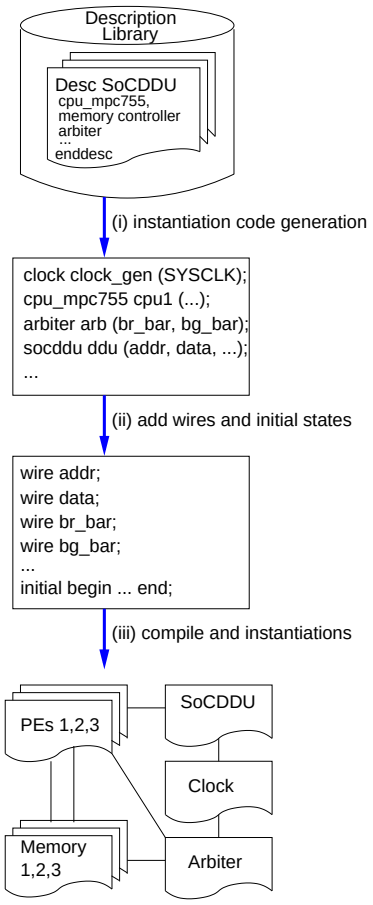


Figure 48: HDL top file generation flow of the δ framework.

tool. Figure 49 shows a GUI tool that generates two files, a makefile and a parameter file. A parameter file contains parameters used to automatically generate a user specified hardware deadlock solution out of the DDU, DAU and PBAU. A makefile is used to generate a user specified hardware deadlock solution by processing modifiable deadlock hardware IP library with parameters. Figure 50 illustrates the generation flow. We use "Verilog Pre-Processor (VPP)" to process modules in a modifiable deadlock hardware IP library according to parameters specified by a user. The GUI tool inputs a type of target processor, the number of processes in the target MPSoC and the number of resources. After choosing a deadlock solution, the user clicks the *Generate* button. Then the GUI generates a makefile and a file that contains appropriate parameters for the specified deadlock IP component. After that, by executing the makefile, a target specific deadlock solution (i.e., one of a

DDU, DAU or PBAU) is automatically generated.

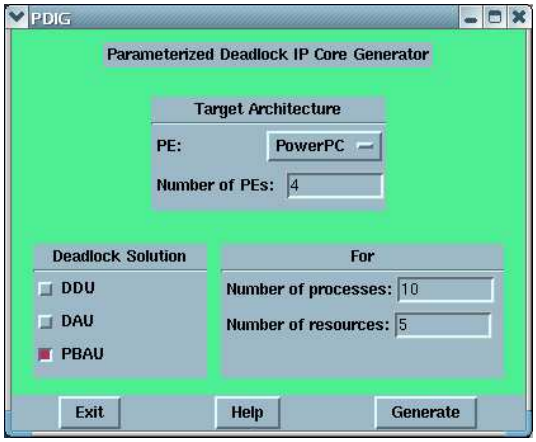


Figure 49: GUI for automatic generation of a hardware deadlock solution.

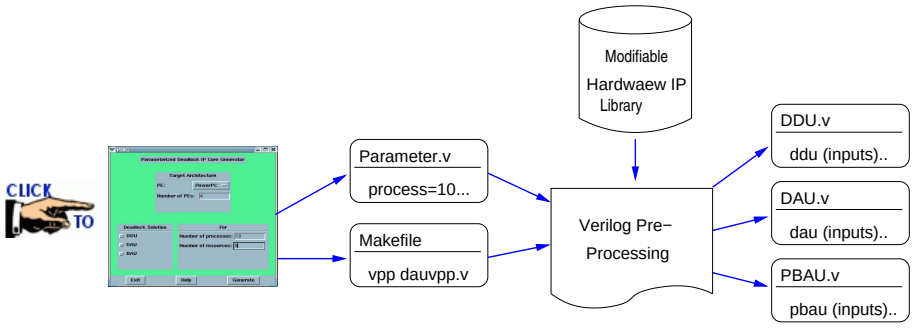


Figure 50: Automatic generation flow of a hardware deadlock solution.

6.4 Summary

This chapter presented the integration of parameterized generation of the DDU, DAU and PBAU into the δ hardware/software RTOS/MPSoC codesign framework that has been used to configure and generate simulatable RTOS/MPSoC designs having both appropriate hardware and software interfaces as well as system architecture. The δ framework is specifically designed to help RTOS/MPSoC designers very easily and quickly explore their design space with available hardware and software modules so that they can efficiently search and discover several optimal solutions matched to the specifications and requirements of their

design before an actual implementation.

This chapter also describes a separate IP generation tool used to automatically generate a hardware deadlock solution out of the DDU, DAU and PBAU according to the numbers of processes and resources that a user specifies.

CHAPTER VII

CONCLUSION

This thesis presents fast and deterministic hardware/software deadlock avoidance methodologies that are easily applicable to real-time multiresource MultiProcessor System-on-a-Chip (MPSoC) design. Our solutions are provided in the form of Intellectual Property (IP) hardware units which we call the Deadlock Avoidance Unit (DAU) and the Parallel Banker's Algorithm Unit (PBAU).

Parallel Deadlock Detection Algorithm (PDDA) and its hardware implementation in the Deadlock Detection Unit (DDU) are proposed by Shui, Tan and Mooney [48]. This thesis illustrates detailed descriptions of PDDA as well as DDU with mathematical representations, software implementations of PDDA and extensive experimentation among the DDU, PDDA in software as well as an $O(m \times n)$ deadlock detection algorithm. We proved the correctness of PDDA with five lemmas and four theorems. We also proved that the DDU has a worst case run-time of $2 \times min(m, n) - 3 = O(min(m, n))$ (where m and n are the numbers of resources and processes, respectively) with two corollaries, one lemma and one theorem. Previous algorithms in software, by contrast, have $O(m \times n)$ run-time complexity. The DDU reduces deadlock detection time by 99%, (i.e., 100X) or more compared to software implementations of deadlock detection algorithms. An experiment involving a practical situation that employs the DDU showed that the time measured from application initialization to deadlock detection was reduced by 46% compared to detecting deadlock in software.

The DAU provides very fast and automatic deadlock avoidance in MPSoC with multiple processors and multiple resources. The DAU avoids deadlock by not allowing any grant or request that leads to a deadlock. In case of livelock resulting from an attempt to avoid

deadlock, the DAU asks one of the processes involved in the livelock to release resource(s) so that the livelock can also be resolved. We devised three novel deadlock avoidance algorithms, implemented the algorithms in Verilog Hardware Description Language (HDL), and synthesized them using an automatic synthesis tool. We simulated two synthetic applications that can benefit from the DAU and demonstrated that the DAU reduces the deadlock avoidance time by over 99% *(about 300X)* and achieves in a particular example approximately 40% speedup of application execution time as compared to the execution time of the same application using the same algorithm in software. The MPSoC area overhead due to the DAU is small, under 0.04% in our SoC example.

While the DAU provides automatic deadlock avoidance for single-instance resource systems, PBAU, a hardware implementation of our novel Parallel Banker's Algorithm (PBA), accomplishes fast, automatic deadlock avoidance for multiple-instance resource systems. PBA is a parallelized version of the Banker's Algorithm for a multiple instance multiple resource system, which was proposed by Habermann. We have implemented PBA in Verilog HDL and synthesized it using an automatic synthesis tool. PBAU provides a system with an $O(n)$ run-time complexity deadlock avoidance with a best case run-time of $O(1)$. We demonstrate that PBAU not only avoids deadlock in a few clock cycles (1600X faster than the Banker's Algorithm implemented in software), but also achieves in a particular example a 19% speedup of application execution time over avoiding deadlock in software. The MPSoC area overhead due to PBAU is small, under 0.05% in our candidate MPSoC example.

While our experiments are not industrial strength full product code, nevertheless we expect similar results as MPSoC designs become more commonplace; we predict that our hardware deadlock solutions can potentially help especially in real-time scenarios where at time-critical moments significant transitions involving many releases, requests and grants occur.

To automate the design of hardware deadlock solutions, we also provide an initial approach to an automatic deadlock hardware generation tool that is capable of generating a custom DDU, DAU or PBAU for a user specified combination of resources and processes, so that users can easily and rapidly implement a particular deadlock hardware solution for their target MPSoCs.

Moreover, we have integrated automatic generation of DDU, DAU and PBAU into the δ hardware/software Real-Time Operating System (RTOS) partitioning framework of which the goal is to speed up RTOS/MPSoC codesign. As MPSoC designs become more common, hardware/software codesign engineers face new challenges involving operating system integration. The δ framework is used to configure and generate simulatable RTOS/MPSoC designs having both appropriate hardware and software interfaces as well as system architecture. The δ framework is specifically designed to help RTOS/MPSoC designers very easily and quickly explore their design space with available hardware and software modules so that they can efficiently search and discover several optimal solutions matched to the specifications and requirements of their design prior to any actual implementation.

In summary, we believe that our approaches initiate a paradigm shift in the context of deadlock solutions for multiprocessor multiresource System-on-a-Chip from exclusive use of software to hardware/software partitioned solutions that enable distribution of part of the burden imposed on processors to a low cost, fast hardware IP core. By providing faster and more deterministic deadlock avoidance for such SoCs, our solutions can improve reliability of systems; thus allowing systems to have the much higher levels of concurrency to be demanded in the near future. Furthermore, using the automatic deadlock hardware generation tool, a customized deadlock hardware IP for a particular target system can easily be generated.

REFERENCES

[1] AKGUL, B., *The System-on-a-Chip Lock Cache*. PhD thesis, School of ECE, Georgia Institute of Technology, Atlanta, GA, Spring 2004. http://etd.gatech.edu/theses/available/etd-04122004-165130/.

[2] AKGUL, B., LEE, J., and MOONEY, V. J., "A System-on-a-Chip Lock Cache with task preemption support," *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES'01)*, pp. 149–157, November 2001.

[3] AKGUL, B. and MOONEY, V., "Parlak: Parametrized Lock Cache generator," *Proceedings of the Design Automation and Test in Europe Conference (DATE'03)*, pp. 1138–1139, March 2003.

[4] AKGUL, B. and MOONEY, V. J., "The System-on-a-Chip Lock Cache," *International Journal of Design Automation for Embedded Systems*, vol. 7, no. 1-2, pp. 139–174, September 2002.

[5] AMI Semiconductor. http://www.amis.com/, 2003.

[6] ASICS World Site. http://www.asics.ws/, 2004.

[7] BELIK, F., "An efficient deadlock avoidance technique," *IEEE Trans. on Computers*, vol. 39, no. 7, pp. 882–888, July 1990.

[8] COFFMAN, E., ELPHICK, M., and SHOSHANI, A., "System deadlocks," *Computing Surveys*, vol. 3, pp. 67–78, June 1971.

[9] COUDERT, O., MADRE, J., and FRAISSE, H., "A new viewpoint on two-level logic minimization," *Proceedings of 30th Design Automation Conference (DAC)*, pp. 625–630, June 1993.

[10] DE MICHELI, G., *Synthesis and Optimization of Digital Circuits*. New York, NY: McGraw-Hill, 1994.

[11] DIJKSTRA, E., "Cooperating sequential processes," Tech. Rep. EWD-123, Technological University, Eindhoven, The Netherlands, September 1965.

[12] DVT solutions. http://www.xilinx.com/esp/dvt/cdv/dvt_solutions/ip/dsp/, 2004.

[13] EZPELETA, J., TRICAS, F., GARCIA-VALLES, and COLOM, J., "A banker's solution for deadlock avoidance in FMS with flexible routing and multiresource states," *IEEE Trans. on Robotics and Automation*, vol. 18, no. 4, pp. 621–625, August 2002.

[14] GEBRAEEL, N. and LAWLEY, M., "Deadlock detection, prevention, and avoidance for automated tool sharing systems," *IEEE Trans. on Robotics and Automation*, vol. 17, no. 3, pp. 342–356, 2001.

[15] GOLD, E., "Deadlock prediction: Easy and difficult cases," *SIAM Journal of Computing*, vol. 7, no. 3, pp. 320–336, 1978.

[16] HABERMANN, A., "Prevention of system deadlocks," *Communications of the ACM*, vol. 12, no. 7, pp. 373–377,385, July 1969.

[17] HAVENDER, J., "Avoiding deadlock in multitasking systems," *IBM System Journal*, vol. 7, no. 2, pp. 74–84, 1968.

[18] HENNESSY, J. and PATTERSON, D., *Computer architecture - A quantitative approach*. San Francisco, CA: Morgan Kaufmann Publisher, Inc., 1996.

[19] HOLT, R., "Comments on prevention of system deadlocks," *Communications of the ACM*, vol. 14, no. 1, pp. 36–38, January 1971.

[20] HOLT, R., "Some deadlock properties of computer systems," *ACM Computing surveys*, pp. 179–196, September 1972.

[21] ITRS. The International Technology Roadmap for Semiconductors 2003, http://public.itrs.net/.

[22] KIM, J. and KOH, K., "An O(1) time deadlock detection scheme in single unit and single request multiprocess system," *IEEE TENCON '91*, pp. 219–223, August 1991.

[23] LANG, S., "An extended banker's algorithm for deadlock avoidance," *IEEE Trans. on Software Engineering*, vol. 25, no. 3, pp. 428–432, May 1999.

[24] LEE, J. and MOONEY, V., "Hardware-software partitioning of operating systems: Focus on deadlock detection and avoidance," *IEE Proc. Computers and Digital Techniques*, January 2005.

[25] LEE, J. and MOONEY, V., "A novel O(n) Parallel Banker's Algorithm for system-on-a-chip," *Proceedings of the Asia and South Pacific Design Automation Conference (ASPDAC 2005)*, January 2005.

[26] LEE, J., MOONEY, V., DALEBY, A., INGSTROM, K., KLEVIN, T., and LINDH, L., "A comparison of the RTU hardware RTOS with a hardware/software RTOS," *Proceedings of the Asia and South Pacific Design Automation Conference (ASPDAC 2003)*, pp. 683–688, January 2003.

[27] LEE, J. and MOONEY, V. J., "An O(min(m,n)) Parallel Deadlock Detection Algorithm," Tech. Rep. GIT-CC-03-41, College of Computing, Georgia Institute of Technology, Atlanta, GA, September 2003.

[28] LEE, J., RYU, K., and MOONEY, V., "A framework for automatic generation of configuration files for a custom RTOS," *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA'02)*, pp. 31–37, June 2002.

[29] LEIBFRIED, T., "A deadlock detection and recovery algorithm using the formalism of a directed graph matrix," *Operation Systems Review*, pp. 45–55, April 1989.

[30] LI, C., XIAO, L., YU, Q., VENKATESAN, R., and GILLARD, P., "Design of a pipelined DSP processor - MUN DSP2000," *Proceedings of Newfoundland Electrical and Computer Engineering Conference (NECEC 2000)*, November 2000.

[31] MAEKAWA, M., OLDHOEFT, A., and OLDEHOEFT, R., *Operating Systems - Advanced Concepts*. Menlo Park, CA: Benjamin/Cummings Publishing Company, 1987.

[32] MCCLUSKEY, E., "Minimization of boolean functions," *Bell System Technical Journal*, vol. 35, pp. 1417–1444, 1959.

[33] Mentor Graphics, Hardware/Software Co-Verification: Seamless. http://www.mentor.com/seamless/, 2004.

[34] Mentor Graphics, XRAY Debugger. http://www.mentor.com/xray/, 2004.

[35] MOONEY, V., *Hardware/software partitioning of operating systems in the book Embedded Software for SoC edited by A. Jerraya, S. Yoo, D. Verkest and N. Wehn*. Boston, MA: Kluwer Academic Publishers, 2003.

[36] MOONEY, V. and BLOUGH, D., "A hardware-software real-time operating system framework for SOCs," *IEEE Design and Test of Computers*, pp. 44–51, Nov.-Dec. 2002.

[37] MOONEY, V. and LEE, J., *Hardware-software partitioning of operating systems: focus on deadlock detection and avoidance edited by IEE*. England: IEE Press, 2005.

[38] MORGAN, S., "Jini to the rescue," *IEEE Spectrum*, vol. 37, no. 4, pp. 44–49, April 2000.

[39] QualCore Logic. http://www.qualcorelogic.com/, 2004.

[40] REEVES, G., "What really happened on mars," *RISKS Forum*, vol. 19, no. 54, January 1998.

[41] RYU, K., *Automatic generation of bus systems*. PhD thesis, School of ECE, Georgia Institute of Technology, Atlanta, GA, Summer 2004. http://etd.gatech.edu/theses/available/etd-07122004-121258/.

[42] RYU, K. and MOONEY, V., "Automated bus generation for multiprocessor SoC design," *Proceedings of the Design Automation and Test in Europe Conference (DATE'03)*, pp. 282–287, March 2003.

[43] RYU, K. and MOONEY, V., "Automated bus generation for multiprocessor SoC design," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 23, no. 11, pp. 1531–1549, November 2004.

[44] SHA, L., RAJKUMAR, R., and LEHOCZKY, J., "Priority inheritance protocols: An approach to real-time synchronization," *IEEE Trans. on Computers*, vol. 39, no. 9, pp. 1175–1185, September 1990.

[45] SHALAN, M., *Dynamic memory management for embedded real-time multiprocessor system-on-a-chip*. PhD thesis, School of ECE, Georgia Institute of Technology, Atlanta, GA, Fall 2003. http://etd.gatech.edu/theses/available/etd-11252003-131621/.

[46] SHALAN, M. and MOONEY, V., "Hardware support for real-time embedded multiprocessor system-on-a-chip memory management," *Proceedings of the Tenth International Symposium on Hardware/Software Codesign (CODES'02)*, pp. 79–84, May 2002.

[47] SHALAN, M., SHIN, E., and MOONEY, V., "DX-Gt: Memory management and crossbar switch generator for multiprocessor system-on-a-chip," *Proceedings of the 11th Workshop on Synthesis And System Integration of Mixed Information technologies (SASIMI'03)*, pp. 357–364, April 2003.

[48] SHIU, P., TAN, Y., and MOONEY, V. J., "A novel parallel deadlock detection algorithm and architecture," *Proceedings of the 9th International Workshop on Hardware/Software Co-Design (CODES'01)*, pp. 30–36, April 2001.

[49] SHOSHANI, A. and COFFMAN, E., "Detection, prevention and recovery from deadlocks in multiprocess, multiple resource systems," *4th Annual Princeton Conference on Information Sciences and System*, March 1970.

[50] STALLING, W., *Operating Systems*. Upper Saddle River, New Jersey: Prentice Hall, 2001.

[51] SUN, D., BLOUGH, D., and MOONEY, V. J., "Atalanta: a new multiprocessor RTOS kernel for system-on-a-chip applications," Tech. Rep. GIT-CC-02-19, College of Computing, Georgia Institute of Technology, Atlanta, GA, March 2002.

[52] Synopsys, Design Compiler. http://www.synopsys.com/products/logic/logic.html, 2004.

[53] Synopsys, VCS Verilog Simulator. http://www.synopsys.com/products/simulation/simulation.html, 2004.

[54] Texas Instruments, TMS320C80. http://focus.ti.com/docs/prod/folders/print/tms320c80.html., 2004.

[55] The MPEG Home Page. http://www.chiariglione.org/mpeg/, 2004.

[56] The Official Bluetooth Website. http://www.bluetooth.com/, 2004.

[57] Xilinx. http://www.xilinx.com/, 2004.

VITA

Jaehwan Lee was born in Daegu, South Korea. After obtaining his master's degree in electrical engineering at Kyeong-book National University, he joined the Agency for Defense Development where he was involved in the design of firing control systems for guided missile systems. While he was working as a digital circuit designer and senior researcher in the industry, a strong desire to teach was growing inside him. Since he deeply respected the teaching and mentoring of students, he was willing to quit a very good job, making an extraordinary decision to come to Georgia Tech and learn enough about computer-aided VLSI system design to hopefully be able to stimulate the curiosity of students and help them efficiently prepare for a career in the field. Now he wishes to embark on becoming an excellent professor as well as a mentor.

Entering Georgia Tech, he joined Georgia New Seoul Baptist Church, where he met many invaluable Christian brothers and sisters, who influenced him tremendously. He believes that God is helping him and prays for a successful accomplishment of his vision realizing the goal of teaching and researching at an American university. Therefore, he now pursues becoming an excellent professor at a university wherever God would direct in the United States or elsewhere.